Contributions à la certification des calculs dans R : théorie, preuves, programmation

Assia Mahboubi

UNIVERSITÉ de NICE-SOPHIA ANTIPOLIS – UFR SCIENCES

École Doctorale STIC

THÈSE

pour obtenir le titre de

Docteur en SCIENCES

de l'Université de Nice Sophia Antipolis

Spécialité : **INFORMATIQUE**

présentée et soutenue par

Assia Mahboubi

Contributions à la certification des calculs dans $\mathbb{R}$: théorie, preuves, programmation

Thèse dirigée par **Loïc Pottier**

Préparée à l'INRIA Sophia Antipolis, projet MARELLE

Soutenue publiquement le **16 novembre 2006** devant le jury composé de :

M.	**André**	**Hirschowitz**	Professeur, Nice-Sophia Antipolis	Examinateurs
M.	**Benjamin**	**Werner**	Chargé de recherche à l'INRIA	
M.	**Loïc**	**Pottier**	Chargé de recherche à l'INRIA	Directeur
M.	**Thierry**	**Coquand**	Professeur, Göteborg	Rapporteurs
M.	**John**	**Harrison**	Chercheur Intel	
Mme	**Marie-Françoise**	**Roy**	Professeure, Rennes I	

Remerciements

Merci à Loïc Pottier qui, après m'avoir fait découvrir les mathématiques dites formelles, a rendu cette thèse possible en acceptant de l'encadrer.

Merci à Marie-Françoise Roy pour sa confiance, sa disponibilité, et son hospitalité dans la tour de maths de Rennes.

Merci à Thierry Coquand et Marie-Françoise Roy pour avoir accepté de rapporter cette thèse. Je suis extrêmement honorée que chacun d'entre eux ait accepté de relire ce travail.

Thanks to John Harrison for having accepted to report on this manuscript despite its being written in french. Thanks also for the numerous suggestions and comments on this work, and especially for the nice unexpected discussions in European trains.

Merci à André Hirschowitz pour avoir accepté de présider ce jury.

Merci à Benjamin Werner pour avoir accepté de faire partie de ce jury, pour ses encouragements, son intérêt pour ce travail et ses suggestions pour son prolongement.

Merci à Benjamin Grégoire pour avoir accepté de partager un bureau avec moi pendant deux ans. L'influence de cette cohabitation sur ce travail est immense. Merci pour ta rigueur, ton enthousiasme, ta patience, tout ce que j'ai appris pendant ces deux ans, bref, merci pour tous ces bons moments.

Merci à tous les membres des équipes Marelle et Everest pour la chaleureuse ambiance qu'ils créent à Sophia.

Merci à Nathalie Bellesso pour son aide matérielle, logistique, administrative, sa gentillesse et sa disponibilité à toute épreuve.

Merci à Daniel Hirschkoff, parce que lui sans lui, je n'en serais sûrement pas là, mais surtout parce que lui non plus il n'aime pas le poisson.

Merci à Hervé et David, pour tout, et surtout pour leur précieuse amitié.

Merci aux copains et à mes parents, ma soeur, pour être là même quand je ne le suis pas.

Merci Erwan.

Table des matières

Introduction

Décidabilité et complexité de l'arithmétique réelle

L'arithmétique réelle du premier ordre a la propriété remarquable d'être *décidable* (Tarski 1951). Ceci signifie que l'on peut écrire une suite d'instructions, un algorithme, qui permet de décider la vérité d'une formule close quelconque de cette théorie. On peut formuler autrement cette propriété en disant que l'on sait construire par un procédé automatique une preuve de toute formule close de la théorie du premier ordre de l'arithmétique réelle, c'est à dire toute formule qui est une combinaison d'inégalités et d'égalités polynomiales quantifiées à coefficients rationnels, sans paramètres.

Au premier abord, un tel résultat semble être de toute première importance pour la branche des mathématiques qui étudie les objets définis par des inégalités polynomiales réelles, à savoir la géométrie algébrique réelle. Par exemple l'existence d'un tel algorithme permet en théorie de déterminer les configurations de variétés algébriques réalisant un ensemble de contraintes sur leur degré, nombre de variables...

En fait, la perspective d'obtenir de nouveaux résultats grâce à cette méthode, et à son implémentation sur un ordinateur, est limitée par la complexité de ce problème de décision. La complexité de l'algorithme historique donné par A. Tarski est énorme : c'est une tour d'exponentielle de hauteur le nombre de variables quantifiées. L'introduction de l'algorithme de Décomposition Algébrique Cylindrique (CAD pour Cylindrical Algebraic Decomposition) par G.E. Collins (Collins 1975) représente une amélioration drastique puisque celui-ci est "seulement" doublement exponentiel en le nombre de variables et polynomial en le degré des polynômes. De plus, cet algorithme est un outil puissant pour l'étude des variétés semi-algébriques, qui donne bien plus d'information sur une famille de polynômes que la réponse au problème de décision. Les recherches ultérieures sur cette procédure de décision (Grigor'ev 1988; Heintz, Roy, et Solernó 1990) ramèneront sa complexité en une double exponentielle en le nombre de blocs de quantificateurs, mais ceci est encore trop élevé pour en faire un outil de calcul formel vraiment performant. À notre connaissance, il n'existe pas à ce jour d'implémentation de ces derniers algorithmes. Les implémentations existantes de l'algorithme de CAD peuvent bien sûr être utilisées pour la résolution de problèmes non triviaux, mais ils n'ont pas la puissance promise par le résultat théorique de décidabilité.

Pourquoi faire des preuves formelles

En marge des systèmes de calcul formel, une autre utilisation de l'ordinateur comme auxiliaire à l'activité du mathématicien a émergé au cours des quarante dernières années

du fait de l'apparition des logiciels dits de preuve formelle. L'implémentation de procédures de décision, même de coût élevé, est un aspect crucial du développement de tels systèmes.

Les premiers systèmes conçus pour faire vérifier des propriétés logiques par un ordinateur se divisent grossièrement en deux familles, selon le rôle joué par la machine. La première est celle des *prouveurs automatiques*. Elle regroupe des systèmes basés sur des méthodes de résolution/unification initiées par les travaux de Herbrand dans les années 1930 puis de J. A. Robinson dans les années 1960. Ces systèmes permettent d'obtenir (lorsque c'est possible étant donnée l'indécidabilité du problème) par des procédés automatiques des preuves d'énoncés de la logique propositionnelle classique ou de la logique classique du premier ordre. Le premier système qui peut être rangé dans cette famille est peut-être le *piano logique*[1] construit en 1869 par W. S. Stanley. Il s'agit d'un système mécanique construit pour des raisons en partie pédagogiques, qui pouvait calculer à partir d'un ensemble de formules booléennes contenant au plus quatre variables distinctes quelles combinaisons sont impossibles.

Une deuxième famille est formée par les *assistants à la preuve interactifs*. Dans ces systèmes, l'utilisateur n'attend pas de la machine qu'elle établisse automatiquement la validité de formules, et d'établir de "nouveaux théorèmes". Au contraire, la fonction de ces systèmes est de vérifier scrupuleusement que la preuve d'un énoncé proposée par l'utilisateur est bien formée, et que par suite l'utilisateur a bien démontré un nouveau théorème. C'est donc en définitive l'utilisateur qui est l'architecte de la preuve, l'ordinateur n'ayant qu'une fonction de vérification. Un exemple historique de ce type d'outils est le système AUTOMATH (de Bruijn 1970). Le système Coq (Coq development team 2004; Bertot et Casteran 2004) fait lui aussi partie de cette famille.

Les premières implémentations de ces systèmes, qu'ils soient automatiques ou interactifs, sont souvent motivées par des applications industrielles comme la modélisation de circuits ou bientôt la preuve de spécifications de programmes. Ce champ d'application des méthodes formelles au sens large est toujours un domaine actif de recherches, que ce soit pour garantir la fiabilité d'application sensibles (aéronautique, médecine,...) ou bien la correction de calculs de bas niveau (circuits, arithmétique flottante,...).

Les motivations d'un mathématicien

Cette motivation initiale rend en pratique nécessaire la formalisation de mathématiques dans de tels systèmes. Dès les années 1970, on assiste à des développements importants qui participent de cet effort, comme la traduction dans le système AUTOMATH du célèbre ouvrage de L. D. Landau *Grundlagen der Analysis* par L. S. Jutting. La vérification par ordinateur d'énoncés mathématiques est un champ de recherche autonome. Le géomètre algébriste C. Simpson a récemment décrit avec enthousiasme l'impact possible de la mécanisation de la vérification de mathématiques par un ordinateur (Simpson 2004). Il relève en particulier huit raisons pour lesquelles à ses yeux "il est une erreur de penser que la preuve formelle sur ordinateur restera toujours à la traîne derrières les mathématiques usuelles. Viendra le moment où les avantages de cette pratique en compenseront les difficultés au point que les mathématiciens utilisant ces outils feront un bond

[1] Le "Logic Piano" est conservé au Museum of the History of Science, Oxford (Royaume-Uni).

en avant." Les arguments qu'il pointe sont variés. En particulier, on peut citer la perspective de changer profondément le mode d'arbitrage des revues scientifiques, qui garantit la fiabilité des résultats mathématiques publiés. La création d'un socle de théories formalisées permettra au lecteur de s'appuyer avec confiance sur des énoncés établis, et vérifiés dans un contexte sans ambiguïté, sans devoir nécessairement investir le temps nécessaire pour acquérir l'expertise d'un domaine mathématique connexe à ses propres intérêts. Les preuves qui mêlent les points de vues de différents horizons mathématiques sont en effet notoirement longues et délicates à vérifier. D'autre part, l'utilisation croissante de calculs, réalisés par des programmes, comme ingrédient de preuves mathématiques exige la mise en œuvre d'un protocole de vérification formelle qui semble bien relever du champ d'application des systèmes de preuve sur ordinateur.

Comment fait-on des preuves formelles

Nous sommes jusqu'à maintenant restés très vagues sur le sens attribué à l'expression "formaliser". De fait il existe de nombreuses façons d'envisager d'utiliser un ordinateur comme support pour stocker, concevoir et vérifier des mathématiques : traitement de texte scientifique, systèmes de calcul formel, bases de données, prouveurs automatiques,... F. Wiedijk a recensé près de 300 systèmes[2] avec lesquels on peut "faire des mathématiques sur ordinateur".

Le cas du système Coq

Le travail que nous présentons ici a été effectué dans le système Coq. Il s'agit d'un système de preuve interactif, qui repose sur le *Calcul des Constructions Inductives* (Coquand et Huet 1988; Coquand et Paulin-Mohring 1990). Ce formalisme est une théorie des types, c'est à dire un λ-calcul typé, qui permet de représenter à la fois des énoncés et des preuves de ces énoncés comme des objets du langage. Cette expressivité est donnée par *l'isomorphisme de Curry-de Bruijn-Howard* (Howard 1980) qui établit une correspondance entre programmation et démonstration : les types du λ-calcul sont vus comme des énoncés de théorèmes et les λ-termes, qui sont des programmes, comme des preuves. Dans un tel système, prouver un théorème consiste à former le type qui correspond à son énoncé, puis à fournir un λ-terme qui a ce type. L'assistant à la preuve est un vérificateur de types (*type checker*) qui permet d'assurer la validité de ce jugement de typage. Un énoncé faux sera donc représenté par un type dont il n'est pas possible de construire un habitant et un énoncé prouvé est associé à un témoin de cette preuve : le λ-terme construit avec le type attendu.

Les systèmes construits sur l'isomorphisme de Curry-de Bruijn-Howard sont les héritiers du système AUTOMATH, conçu par N. de Bruijn. Ce dernier a formulé ce qui est maintenant connu comme le *critère de de Bruijn* pour les assistants à la preuve. Le second théorème d'incomplétude de Gödel exclut la possibilité pour un assistant à la preuve de vérifier la cohérence de la logique qu'il implémente. Il est néanmoins peu satisfaisant de faire reposer la validité de théorèmes sur un programme, qui comporte potentiellement des erreurs. Un assistant à la preuve gagne de fait en fiabilité, s'il est constitué d'un

[2] http ://www.cs.ru.nl/~freek/digimath/bycategory.html

programme relativement court et simple, à l'échelle de la relecture par un être humain, qui constitue le vérificateur de preuves. Toute preuve doit alors consister en la production d'un objet témoin, qui est vérifié par ce programme. Ce programme vérificateur de preuve est appelé le noyau et dans le cas d'un système comme Coq, il s'agit en fait du type-checker. Ce n'est pas la seule façon de mettre en œuvre le critère de de Bruijn : on peut par exemple choisir comme objet de preuve les dérivations de preuve, comme c'était le cas des premières versions du système HOL.

Un autre aspect remarquable d'un système basé sur la théorie des types comme le système Coq est qu'il possède de par le λ-calcul qu'il implémente un véritable langage de programmation. Dans le cas du Calcul des Constructions Inductives, le mécanisme d'évaluation de ce langage est donné gratuitement par les règles de réduction.

Définir, programmer, prouver en Coq, un exemple

Un exemple simple permet de mettre en situation ces différents aspects du système. Dans le système Coq, l'utilisateur a la possibilité de donner des définitions inductives en précisant la liste des règles de bonne formation des éléments d'un type. Ces règles sont appelées les *constructeurs du type inductif*. Par exemple, les entiers de Péano sont représentés par le type `nat` du code 1. Il s'agit du plus petit type qui contienne la constante déclarée `0` et qui soit clos par l'opération `S`. Lorsque l'utilisateur déclare ce type[3], le système matérialise cette propriété de "plus petit type" en générant automatiquement les principes d'induction associés.

```
Coq< Inductive nat : Set :=
    O : nat
   |S : nat → nat.

nat is defined
nat_rect is defined
nat_ind is defined
nat_rec is defined
```

Code 1: *Entiers de Péano en Coq*

Le principe `nat_rec` permet de définir des fonctions par filtrage sur un entier et le principe `nat_ind` de faire des raisonnements par récurrence sur un entier[4] :

```
nat_ind
     : ∀ P : nat → Prop,
       P O → (∀ m : nat, P m → P (S m)) → ∀ m : nat, P m
```

On peut maintenant définir une fonction sur ces entiers, qui ajoute 2 à son argument :

```
Definition plus2(n:nat):= S (S n).
```

On peut *évaluer* cette fonction sur un argument donné en effectuant la β-réduction.

```
Coq < Eval compute in (plus2 8).
     = 10
```

[3] en fait défini dans la bibliothèque standard du système.

[4] Le système génère un principe par *sorte* : **Prop**, **Set** et **Type**.

En fait l'instruction **compute** n'est pas limitée à la β-réduction. Elle permet par exemple également de réduire les opérations définie par récurrence, qui sont des opérations de *point fixe*. Le système Coq permet en effet de définir des fonctions par récursion dite structurelle sur un argument dont le type est inductif. La récursion est dite structurelle si elle est repose sur le principe de récursion généré au moment de la définition du type inductif. Par exemple, la fonction qui calcule le minimum de deux entiers peut se programmer de la façon suivante :

```
Fixpoint min n m {struct n} : nat :=
  match n, m with
  | O, _ ⇒ 0
  | S n', O ⇒ 0
  | S n', S m' ⇒ S (min n' m')
  end.
```

Dans le code ci-dessus, on indique explicitement l'argument sur lequel porte la récursion grâce au mot-clef **struct**. On peut toujours *réduire* l'application d'une telle fonction définition par récursion grâce à la tactique **compute**.

```
Coq < Eval compute in (min 3 8).
     = 3
     : nat
```

Lorsqu'on veut faire une preuve par récurrence sur un objet défini par induction, le système utilise par défaut le principe généré au moment de la définition du type. On va voir sur un exemple comment on peut manipuler un type inductif dans une preuve. La propriété de commutativité s'énonce ainsi :

```
Lemma min_comm : ∀ n m, min n m = min m n.
```

La preuve Coq de ce lemme, qui est une construction de λ-terme, se fait de manière interactive. L'utilisateur donne au système une succession d'instructions, appelées *tactiques*. Ces instructions font évoluer le but à prouver, en créant éventuellement de nouveaux sous-buts, jusqu'à ce que la preuve soit complètement achevée. Elles sont séparées par des points. Dans tout cet exemple, à gauche se trouvent les instructions tapées par l'utilisateur et à droite la réponse du système Coq. La preuve se fait par récurrence sur n, le premier argument, puis, pour le cas de base ainsi que pour le cas récursif, on raisonne par cas sur le deuxième entier m.

```
Lemma min_comm : ∀ n m,
                 min n m = min m n.

Proof.
  induction n.
```

```
2 subgoals

  ============================
 ∀ m : nat, min 0 m = min m 0

subgoal 2 is:
 ∀ m : nat,
      min (S n) m = min m (S n)
```

FIG. 1: *Récurrence sur le premier argument*

La première étape, sur la figure 1, montre les deux sous-buts générés après l'instruction qui initie une récurrence sur l'entier `n`.

Ensuite, pour prouver le cas de base, on raisonne par cas sur `m`, ce qui génère les deux sous-cas de la figure 2. La tactique **`destruct`** détaille les cas possibles, c'est à dire les constructeurs possibles, pour un élément du type `nat`, sans opérer de récurrence.

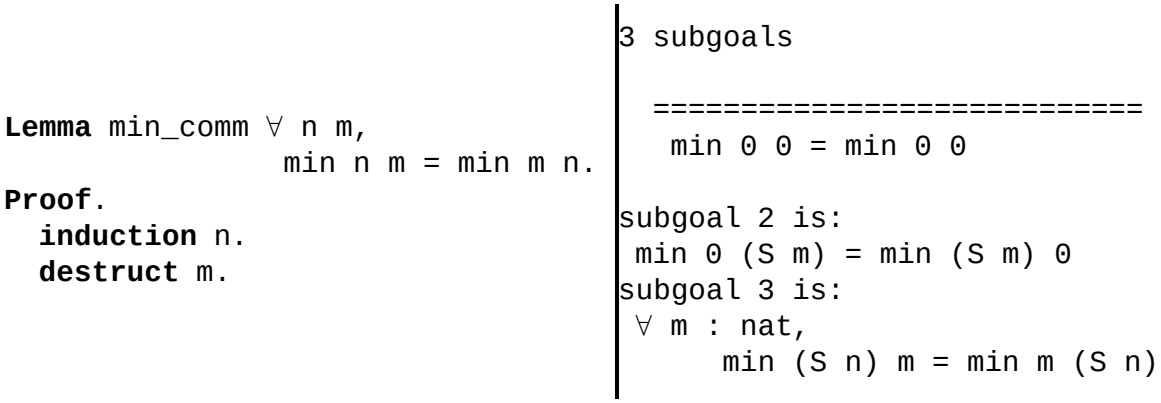


FIG. 2: *Par cas sur le deuxième argument*

On peut maintenant conclure le premier sous but en utilisant la réflexivité de la relation d'égalité. Le sous-but à résoudre devient le deuxième cas de l'argument `m`, comme représenté sur la figure 3.

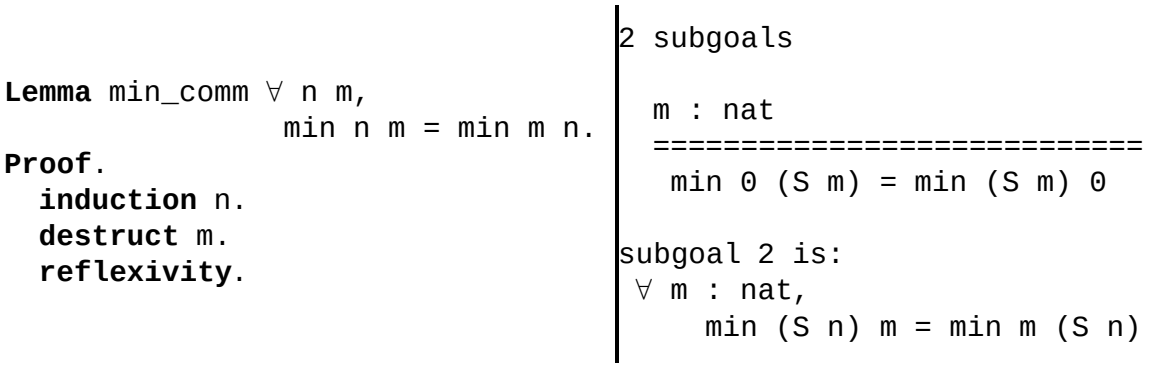


FIG. 3: *Le premier sous-cas est résolu*

Pour clore le deuxième sous-but, il faut réduire la fonction `min`, afin d'obtenir la valeur `0` de part et d'autre du signe `=` et conclure par réflexivité de l'égalité. Ceci termine le cas de base de la récurrence et l'utilisateur doit maintenant prouver le cas récursif de la récurrence sur `n`. L'hypothèse de récurrence a été générée et placée dans le contexte des hypothèses disponibles, comme représenté sur la figure 4.

```
Lemma min_comm ∀ n m,
                min n m = min m n.
Proof.
  induction n.
  destruct m.
  reflexivity.
  simpl min. reflexivity.
```

```
1 subgoal

  n : nat
  IHn : ∀ m : nat,
             min n m = min m n
  ============================
   ∀ m : nat,
        min (S n) m = min m (S n)
```

FIG. 4: *Le deuxième sous-cas est résolu*

On raisonne une deuxième fois par cas sur m, et comme précédemment, le premier cas se résout en évaluant la fonction `min`. On doit enfin traiter le dernier sous-cas qui se présente comme en figure 5.

```
Lemma min_comm ∀ n m,
                min n m = min m n.
Proof.
  induction n.
    destruct m.
    reflexivity.
    simpl min. reflexivity.
      destruct m.
      simpl min. reflexivity.
```

```
1 subgoal

  n : nat
  IHn : ∀ m : nat,
             min n m = min m n
  m : nat
  ============================
 min (S n) (S m) = min (S m) (S n)
```

FIG. 5: *Le deuxième sous-cas est résolu*

Cette fois l'évaluation de la fonction `min` va transformer le but à prouver en

```
S (min n m) = S (min m n)
```

qui se prête à l'utilisation de l'hypothèse de récurrence. La figure 6 représente le script complet.

Cette suite d'instructions a raffiné progressivement un λ-terme à trous, qui est complètement formé lorsque le système affiche le message **`Proof completed`**. Ce terme doit avoir le type annoncé par l'énoncé du lemme.

Pour enregistrer ce résultat dans la base des énoncés prouvés, il faut faire vérifier le λ-terme construit par le type-checker de Coq. Si cette vérification établit que le λ-terme représentant la preuve a bien le type de l'énoncé, alors Coq définit un nouvel objet dans son contexte, nommé `min_comm` et l'utilisateur pourra désormais utiliser ce résultat dans ses preuves ultérieures. Cette opération de vérification et de sauvegarde est effectuée par l'instruction **`Qed`**, pendant de l'instruction **`Proof`** qui ouvre la preuve.

La récursion structurelle n'est pas le seul moyen de définir des opérations par point fixe dans le système Coq. Pour définir des fonctions par récurrence en utilisant un principe différent de celui qui a été généré automatiquement, on devra en général utiliser la technique dite de *récursion bien fondée* en donnant explicitement au système une preuve que l'ordre sur lequel repose la définition est bien fondé.

```
Lemma min_comm ∀ n m,                    |
                min n m = min m n.       |
Proof.                                   |
  induction n.                           |
    destruct m.                          |
      reflexivity.                       | Proof completed.
        simpl min. reflexivity.          |
    destruct m.                          |
      simpl min. reflexivity.            |
        simpl min. rewrite (IHn m).      |
        reflexivity.                     |
```

FIG. 6: *Fin de la preuve*

La place du calcul

Le calcul possède ainsi un statut *primitif* dans le Calcul des Constructions Inductives, statut fortement illustré par la règle de typage dite de *conversion*. La relation de conversion, notée $\equiv$, est une relation d'équivalence qui identifie deux termes "égaux modulo le calcul", dans un sens plus large que la simple β-réduction : elle englobe aussi d'autres règles comme les réductions de points fixes ou le pliage/dépliage de définitions intermédiaires. La règle (de typage) de conversion indique que deux types qui sont convertibles possèdent les mêmes habitants :

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s \qquad T \equiv U}{\Gamma \vdash t : U}$$

Cette règle a des conséquences fondamentales sur la forme des preuves dans le Calcul des Constructions. Par exemple, pour donner une preuve de la proposition `1 + 2 = 3`, dont l'énoncé est un type de Coq, il faut construire un λ-terme dont le type est bien `1 + 2 = 3`. Or les termes `1 + 2` et `2` sont convertibles, car l'opération d'addition appliquées aux arguments `1` et `2` *s'évalue* en `3`. Par conséquent, les types `1 + 2 = 3` et `3 = 3` dont convertibles. Ainsi pour prouver que `1 + 2 = 3`, d'après la règle de conversion, il suffit de donner un terme de type `3 = 3`, qui est en Coq le terme `(refl_equal 3)`.

Égalité, relations d'équivalences

Le traitement de la relation d'égalité en théorie des types est un sujet délicat. La relation d'égalité que nous avons utilisée ci-dessus est celle qui est primitive au système. Il s'agit d' une *égalité syntaxique*, définie comme la plus petite relation réflexive.

```
Inductive eq (A : Type) (x : A) : A → Prop :=  refl_equal : x = x.
```

Cette définition indique qu'un terme ne peut être égal qu'à ... lui-même exactement, ou plutôt comme on l'a vu, à un terme qui lui est convertible.

Cette définition est assez éloignée de la pratique des mathématiciens, qui utilisent assez souvent implicitement une relation d'équivalence comme relation d'égalité. Il est par exemple très rare que le symbole d'égalité entre deux fractions différentes de même

forme irréductible ne soit pas le symbole = et ce sans que cela soulève de problème particulier dans l'esprit du lecteur. De la même façon, c'est souvent le contexte dans lequel on le rencontre qui permet de déterminer la nature d'un symbole d'égalité entre deux fonctions : par exemple on peut avoir affaire à une égalité syntaxique, mais aussi à une égalité point par point de l'intersection des domaines de définition, ou encore à une égalité des valeurs point par point *et* des domaines de définitions.

Cette définition est encore plus éloignée de l'égalité des systèmes de calcul formel dont la sémantique est parfois très floue (Harrison et Théry 1998). La manipulation de structures quotients en théorie des types a fait l'objet de nombreux travaux, et demande une vigilance particulière pour maintenir la cohérence du système (Chicli, Pottier, et Simpson 2002).

Dans le système Coq, lorsque la relation d'équivalence utilisée n'est pas l'égalité de Leibniz, on doit toujours déclarer explicitement une structure de quotient. Le système possède un mécanisme qui permet la manipulation de telles classes d'équivalences, et en particulier permet d'effectuer des *réécritures* : il s'agit de la notion de sétoïde (Hofmann 1995; G. Barthe et Pons. 2003). Un sétoïde est simplement la donnée d'un type et d'une relation d'équivalence sur ce type. Toutefois, pour pouvoir réécrire des égalités exprimée grâce à cette relation d'équivalence sous un symbole de fonction, il faut montrer que cette fonction passe au quotient pour la relation d'équivalence, on dit alors qu'il s'agit d'un *morphisme* pour la structure de sétoïde. Il est alors techniquement possible de faire des preuves par réécriture de ces relations d'équivalence mais ce procédé, et en particulier la recherche d'occurrences, donne lieu à des problèmes d'efficacité qui peuvent devenir un problème rédhibitoire dans une formalisation. Le système de sétoïdes de Coq a récemment été refondu et amélioré (Sacerdoti Coen 2006), mais reste en cours de développement.

Polymorphisme, types dépendants

Le dernier trait du système Coq que nous évoquerons dans cette introduction concerne son système de types et en particulier l'usage qu'il permet des *types dépendants*.

Le λ-calcul est un formalisme destiné à représenter des fonctions. Néanmoins comme les seules règles de construction de ses termes sont l'abstraction, qui crée une nouvelle fonction, et l'application, qui applique une fonction à un argument, ce système représente *beaucoup* de fonctions, en particulier, il en représente trop pour servir à modéliser la logique. Si on formalise un prédicat par un terme du λ-calcul qui s'applique à un argument comme le prédicat s'applique à un objet, on peut encoder dans ce système le paradoxe de Russell.

Pour restreindre la classe d'objets considérée et ainsi se prémunir de tels paradoxes, on introduit un système de types donné par des règles qui décrivent en fait les conditions de bonne formation des fonctions acceptées par le système. Les choix des règles de typage contraignent ainsi du même coup l'expressivité du système logique correspondant par l'isomorphisme de Curry-de Bruijn-Howard. Définir une fonction en donnant son graphe revient à décrire une partie du *produit* cartésien de son domaine de définition et de son codomaine. Les types des abstractions, c'est à dire des fonctions mathématiques, sont par suite appelés *types produits*.

Dans le λ-calcul *simplement typé*, la seule opération qui permet de construire de nouveaux types est la *flèche* : on dispose d'un ensemble de types atomiques et puis si

A et B sont deux types, on peut former le nouveau type $A \to B$, qui est le type des fonctions prenant comme argument un objet de type A et qui calculent un résultat de type B. Les types construits dans ce formalisme correspondent exactement aux énoncés de la *logique propositionnelle intuitionniste*.

Pour construire un λ-calcul typé dont la puissance logique dépasse celle du calcul propositionnel, il faut enrichir les constructions de types possibles. Nous ne donnerons pas ici les choix des règles de typage des produits dépendants effectués par le système Coq. La solution adoptée est celle qui offre la puissance maximale en conservant la consistance logique du système[5]. Nous donnons un très bref apperçu des conséquences de ces choix, qui seront largement utilisés dans ce qui suit. Une description plus détaillé des choix du système Coq se trouve dans (Bertot et Casteran 2004).

Le *polymorphisme* introduit plus de souplesse dans la description du support du graphe des fonctions. Il permet en fait une quantification universelle dans les types. Ainsi le terme $\lambda x.x$ qui représente la fonction identité, pourra être typée par $\forall A.A$, qui se lit : "quel que soit le type A de l'argument, cette fonction calcule un résultat de type A". Dans le lambda calcul simplement typé, on ne peut donner un type aussi général : on doit reprogrammer la fonction identité pour chaque type. Le λ-calcul polymorphique a été introduit par J.-Y. Girard avec le système $\mathcal{F}$. Le polymorphisme de Coq est très proche de l'approche qui a contribué au succès des langages fonctionnels de la famille ML. La classe de fonctions typables par ce système augmente beaucoup : elle contient toutes les fonctions dont on peut prouver la totalité en arithmétique du second ordre, ce qui inclue en particulier toutes les fonctions primitives récursives. Du point de vue logique, on a introduit un quantificateur universel dans les formules considérées, de fait le système $\mathcal{F}$ a l'expressivité de l'arithmétique intuitionniste du second ordre.

Dans le système Coq, le système de types autorise des constructions d'*ordre supérieur*. On dispose en fait de règles de typage qui permettent de construire des fonctions dont les arguments peuvent eux-mêmes être fonctionnels. Combiné avec le polymorphisme ce trait du système de typage permet de dépasser la classe des fonctions primitives récursives et de programmer par exemple la fonction d'Ackermann. Du point de vue logique, on manipule des énoncés d'ordre supérieur, dont les quantificateurs peuvent même porter sur des propriétés. Les énoncés des principes `nat_rec` et `nat_ind` présentés ci-dessus donnent un exemple de cette possibilité. La représentation en Coq de l'égalité par l'égalité de Leibniz est également possible grâce aux types d'ordre supérieur. Le type du prédicat `eq` dont on a donné la définition est :

```
eq : ∀ A : Type, A → A → Prop
```

Les types dépendants introduisent quant à eux plus de finesse dans la description donnée par les types. En effet, il est possible de construire des types de données à paramètres grâce à ce principe, puisqu'il autorise l'écriture de programmes qui calculent des types de données. Ainsi il est possible de définir un type de *vecteur* tel que pour tout entier n, les habitants du type *vecteur* n sont les listes de longueur n.

Une des possibilités pour programmer de tels objets est d'utiliser la définition du code 2, qui est celui de la bibliothèque standard du système. Les *vecteurs* sont définis par induction[6], et le type `vector` prend en paramètre un entier `n` qui est la longueur de

[5] ce qui signifie que tous les types du système ne sont pas habités.

[6] Nous n'insistons pas ici sur la possibilité offerte par le Calcul des Constructions Inductive de créer des types inductifs, ce qui n'est pas une construction primitive dans le λ-calcul.

ses habitants.

```
Variable A : Set.
Inductive vector : nat → Set :=
  | Vnil : vector 0
  | Vcons : ∀ (a:A) (n:nat), vector n → vector (S n).
```

Code 2: *Vecteur de longueur n*

Les types dépendants permettent d'inclure dans le type d'un objet une spécification très riche, mais la programmation en Coq avec de telles structures de données nécessite un soin particulier. Rappelons par exemple que l'égalité de Leibniz ne permet de comparer que les objet de même type, or en utilisant la définition du code 2, deux vecteurs de longueur différentes ont un type différent. De même, lorsqu'on veut construire une fonction par filtrage et que le type des résultats de cette fonction est dépendant de l'un de ces arguments, on aura besoin de donner de l'information supplémentaire au système de typage. Par exemple, le code 3 propose une fonction qui renvoie un vecteur d'entiers de longueur n qui ne contient que des zéros.

```
Fixpoint nul_vector(n:nat):(vector nat n):=
  match n return vector nat n with
   |O ⇒ (Vnil nat)
   |S n ⇒ Vcons nat O n (nul_vector n)
  end.
```

Code 3: *Exemple de filtrage dépendant*

La construction **match ... return ... with** permet de donner explicitement après le mot-clef **return** le type du résultat de chaque branche du filtrage en fonction du motif.

Dans le système Coq, il existe un cas particulier de types inductifs, avec un seul constructeur, appelés types d'*enregistrements*. Cet unique constructeur peu être vu comme une fonction qui prend autant d'arguments qu'a de champs l'enregistrement que l'on veut construire et qui calcule un nouvel enregistrement si on lui fournit toutes les valeurs des champs.

Par exemple on peut décrire les points du plans à coordonnées entières, comme dans le code 4, par le type enregistrement est plane_point. L'unique constructeur mk_plane_point prend en argument les valeurs des champs abs et ord pour construire un nouvel enregistrement, qui représente un nouveau point.

```
Record plane_point : Set := mk_plane_point{abs : Z, ord : Z}.
```

Code 4: *Points du plan*

La combinaison de ces types enregistrements avec la dépendance constitue un outil d'abstraction puissant. Dans le code 5 on donne un exemple simple d'enregistrement dépendant qui représente un point à coordonnées entières en dimension n :

```
Record point(n:nat) : Set := mk_point{coord:vector Z n}.
```

Code 5: *Points de l'espace*

Ces enregistrements dépendants représentent une alternative au système de modules de Coq (Chrzaszcz 2003). Bien sûr ces enregistrements n'offrent pas le même confort d'utilisation en particulier dans la gestion des noms, mais il en étendent le champ d'utilisation en permettant notamment des constructions récursives.

Comment fait-on faire des preuves formelles (...par l'ordinateur)

L'utilisation d'un assistant à la preuve interactif comme le système Coq pour formaliser un théorème non trivial de mathématique demande en règle générale un effort considérable, compte tenu du fait que l'utilisateur connaît déjà une preuve "papier" du résultat qui l'intéresse. En effet, il existe une différence de nature significative entre un raisonnement *informel*, intelligible par un mathématicien, et un raisonnement *formel* vérifiable par une machine. Le mathématicien est capable grâce à sa culture et à ses facultés de représentation mentale d'inférer des parties manquantes dans une preuve, en restant convaincu, à tort ou à raison, par sa validité. L'ordinateur quand à lui ne fait preuve d'aucune initiative ni créativité, et le raisonnement formel se décompose en étapes de très bas niveau qui sont les règles de la logique utilisée comme formalisme.

De cette façon, la notion subjective de *preuve élégante* pour un utilisateur de système de preuves formelles est souvent assez éloignée de la conception d'un mathématicien au sens "traditionnel" du terme. Le travail de formalisation de mathématiques à l'aide d'un système de preuves formelle comporte de façon incontournable un travail bibliographique préliminaire pour recenser les démonstrations publiées dans la littérature. Il s'agit en effet de saisir quels arguments sont inévitables pour établir un résultat et quels aspects de la preuve peuvent être aménagés pour un traitement formel. L'utilisateur d'un système comme Coq sera par exemple amené à privilégier lorsque c'est possible une preuve qui peut être ramenée à la correction d'un calcul.

Nécessité de l'automatisation

Néanmoins, pour envisager de façon réaliste un essor des formalisations mathématiques dans un assistant à la preuve, on ne peut compter sur l'effort démesuré que peut représenter la preuve formelle d'énoncés très simples souvent implicitement admis dans une preuve sur papier. De la même façon que les systèmes de calcul formel ont permis de soulager le mathématicien de calculs symboliques pédestres, nécessaires mais sans intérêt propre, on doit exploiter la puissance de calcul de l'ordinateur au mieux pour effectuer de façon automatique les preuves formelles laborieuses qui peuvent être automatisées.

La nécessité d'enrichir les assistants à la preuve interactifs avec des procédures de preuve automatique constitue un enjeu majeur du développement de ces systèmes. Les auteurs de récents développements formels significatifs ont souvent pointé l'aspect crucial de tels outils, que ce soit en soulignant leur rôle déterminant ou bien en soulignant les conséquences de leur insuffisance. En particulier, J. Avigad, K. Donnelly, D. Gray et P. Raff ont insisté sur le travail fastidieux qu'induit le manque d'automatisation de la théorie des réels en décrivant leur travail de mécanisation de la preuve du théorème des nombres premiers dans le système Isabelle (Avigad, Donnelly, Gray, et Raff 2005).

Il existe plusieurs façon de concevoir des programmes qui étendent la puissance d'un assistant à la preuve et leur mise en oeuvre dépend bien sûr du système sur lequel l'assistant est bâti. D'une façon générale, il existe deux grandes familles d'outils de preuve automatiques intégrés à des assistants à la preuve.

L'approche sceptique

La première approche consiste à tirer partie de l'expertise d'un système spécialisé, externe au logiciel de preuve formelle. Par exemple, on peut espérer confier des calculs symboliques à un système de calcul formel, ou encore exploiter la technologie existante des prouveurs automatiques du premier ordre. Malheureusement, cette approche n'est pas forcément la plus simple, ni la plus adaptée au contexte d'un assistant à la preuve. En effet, un programme externe dont on branche la sortie vers le système de preuve est forcément considéré avec scepticisme par ledit système, qui va considérer le programme externe comme un oracle dont il faut vérifier la réponse.

En effet l'oracle fournit le résultat d'un calcul mais pas sa preuve formelle complète. Il s'agit donc de recomposer automatiquement, si c'est possible, une preuve formelle à partir de la réponse de l'oracle. Cette étape peut s'avérer difficile, si l'oracle n'est pas conçu pour fournir une trace suffisante de ses calculs. Elle peut aussi être réalisable mais très coûteuse, tout d'abord en temps, car la trace des calculs que l'on demande à l'oracle peut être très importante, mais aussi en espace, car si le système de preuves formelles doit comme Coq construire un témoin de la preuve, celui-ci peut voir sa taille exploser.

Un autre problème inhérent à cette approche est la différence de sémantique entre les outils de calcul et les outils de preuve formelle (Harrison et Théry 1998), qui peuvent rendre délicat le procédé de preuve automatique des résultats des calculs. En fait, cette approche est bien adaptée à des problèmes dont la solution est potentiellement difficile à trouver, mais pour lesquels il est facile de montrer formellement la correction du résultat.

Un exemple de réussite de cette méthode est la procédure intégrée au système HOL-Light par J. Harrison pour manipuler des systèmes d'inégalités sur les nombres réels. Cette heuristique est basée la décomposition de polynômes positifs en sommes de carrés (Harrison 2005). Elle utilise un oracle basé sur des techniques avancées de programmation semi-linéaire, issues des travaux de P. Parillo. L'outil repose sur des méthodes de calcul sophistiquées mais le résultat de ces calculs est particulièrement simple à exploiter pour un système de preuve : il suffit de vérifier une identité d'anneau et de montrer un lemme qui établit qu'une somme de carrés arbitraire est un polynôme positif.

Dans le système Coq une expérience a été menée pour faire communiquer le système avec le logiciel de calcul formel Maple. Cette contrbution[7] permet d'utiliser le système de calcul formel comme un oracle de calcul qui propose des factorisations, des développements ou des simplifications, en laissant l'utilisateur valider ce calcul par une preuve. Cette preuve peut être obtenue dans les cas favorables par une tactique qui prouve automatiquement des identités algébriques dans les structures de corps. Malheureusement, la combinatoire de cette dernière tactique, qui génère des obligations des preuves d'identités d'anneaux, devient rapidement un facteur limitant dans la taille des expressions que l'on peut traiter avec un tel outil, qui demeurent bien en deçà des capacités de calcul de

[7] http ://coq.inria.fr/contribs-eng.html, D. Delahaye et M. Mayero, 2002

l'oracle Maple. Cette tactique représente néanmoins un outil très utile pour les manipulations symboliques d'anneau, comme par exemple les factorisations de polynômes, que l'on sait prouver efficacement dans le système Coq.

L'approche autarcique

L'approche autarcique consiste à n'utiliser que les ressources propres du système. Les possibilités d'extensions autarciques d'un système de preuves formelles interactif dépendent fortement de la façon dont le système est implémenté.

En toute généralité, une approche autarcique repose uniquement sur la possibilité de décrire un enchaînement d'instructions de l'assistant à la preuve, qui doit produire automatiquement une preuve formelle pour une classe spécifiée d'énoncés. Il s'agit en fait d'écrire un programme, dont les instructions sont des manipulations logiques (applications de règles d'introduction/élimination ...) et syntaxiques (substitutions ...). Le langage qui permet d'écrire un tel programme est appelé *méta-langage* du système de preuves, puisqu'il effectue des opérations qui sortent du système lui-même, à un niveau méta-mathématique. Le méta-langage ML du système LCF, dû à R. Milner est l'ancêtre d'une famille de langage de programmation fonctionnels parmi lesquels le langage Ocaml.

Dans le système Coq, il existe deux versions de méta-langage. Le premier est Ocaml[8], qui est en fait le langage dans lequel est programmé le système. En tant que tel, il permet d'effectuer toutes les opérations du niveau méta-mathématique. Récemment, un deuxième métalangage a été intégré au système, appelé Ltac (Delahaye 2000). Offrant bien sûr beaucoup moins de possibilités de programmation que le langage Ocaml, qu'il ne tend pas à remplacer, il est d'usage léger et permet en particulier de programmer des tactiques en ligne de commande, sans compilation du système.

Il y a également deux manières d'envisager la programmation d'une nouvelle tactique pour le système Coq. Outre celle que nous avons décrite ci-dessus, il est également possible d'exploiter le langage de programmation fourni par la théorie des types qui sous-tend le système pour programmer les calculs nécessaires à une procédure de décision, non nécessairement complète, tel qu'on pourrait le faire dans un langage de programmation arbitraire. En général, ce procédé passe par une description de la classe de problèmes que l'on veut décider à l'aide de types de données plus adaptés au calcul et à des opérations de traduction de l'énoncé concret vers sa représentation abstraite, support du calcul. C'est pourquoi on appelle cette démarche l'*approche à deux niveaux*, ou encore *réflexion*. Ensuite, si l'on peut donner une preuve formelle dans le système de la correction de ces calculs, on peut utiliser ce théorème de correction et la puissance de calcul du système pour donner une preuve formelle des énoncés dans le spectre de la procédure de décision. L'avantage de cette façon de concevoir des tactiques est son efficacité, toutefois subordonnée à celle de l'algorithme de décision, et la maîtrise de la taille du terme de preuve qu'elle permet. D'autre part, les directions de développement récentes du système Coq mettent l'accent sur la puissance de calcul. En particulier, l'introduction depuis la version V8 d'un mécanisme de réduction compilée (Grégoire et Leroy 2002; Grégoire 2003) a été déterminant dans ce sens.

[8] http ://caml.inria.fr/ocaml

Automatisation pour les nombres réels

L'utilisateur qui développe une preuve formelle utilisant des nombres réels dispose dans le système Coq d'une panoplie d'outils de preuve automatique. Par exemple, les égalités modulo la structure d'anneau des réels peuvent être prouvées automatiquement par la tactique `ring`, certaines égalités de la structure de corps peuvent être prouvées automatiquement par la tactique `field`. Déduire une inégalité linéaire d'hypothèses qui sont également des relations d'égalité ou inégalités entre des termes linéaire peut être réalisé automatiquement par la tactique `fourier`, qui applique la méthode de Fourier-Motzkin.

Néanmoins, il reste de nombreux cas dans lesquels l'utilisateur doit passer un temps important à prouver des résultats sans contenu mathématique sophistiqué. Voici un exemple des preuves formelles irritantes qui demandent un temps de travail bien trop long :

```
1 subgoal

  x1 : R
  x0 : R
  v0 : R
  v1 : R
  H : x1 > 0
  H0 : x0 > 0
  H1 : v0 > 0
  H2 : v1 > 0
  H3 : x1 >= v1 + 1
  H4 : x0 >= v0
  ============================
   x1 * x0 + x1 > v0 + (v1 * v0 + v1)
```

Code 6: *Un exemple de but à automatiser*

On ne peut pas utiliser la tactique `fourier` car le but n'est pas linéaire, et la preuve, par ailleurs très facile à écrire sur le papier, repose sur une refactorisation du but et à l'application d'opérations de transitivité entre les différentes hypothèses. Chacune de ces étapes est explicite et demande de connaître, ou de rechercher, le nom du lemme de la bibliothèque qui correspond à la règle de transitivité appliquée, car on peut mélanger inégalités strictes et larges.

En fait, la preuve de l'énoncé décrit sur le code 6 n'est pas si difficile à la main mais prouver ce genre de but au cas par cas n'est pas satisfaisant. D'une part même si les preuves du types de l'énoncé 6, elles peuvent être très nombreuses dans un développement et retarder ainsi inutilement la preuve formelle d'un résultat plus intéressant. Cet exemple particulier est issu de calculs générés par le logiciel CiME qui permet entre autres de calculer automatiquement la terminaison de systèmes de réécriture[9]. Savoir traiter automatiquement ce type de buts permettrait de pouvoir interfacer les deux systèmes et de donner des preuves automatiques et formelles de la terminaison d'ordres de réécriture par interprétation polynomiale. D'autre part, un tel exemple de petite taille demande déjà

[9] http ://cime.lri.fr/

la succession d'au moins une dizaine d'instructions. La taille des scripts de preuve croît très vite avec le nombre d'hypothèses et les arguments nécessaires à la preuve peuvent être plus sophistiqués, utilisant par exemple la monotonie, la convexité ou encore le signe constant des polynômes mis en jeu.

Contributions

La motivation de ce travail est double. Les assistants à la preuve atteignent une maturité qui permet d'envisager des formalisations, de taille conséquente, de résultats mathématiques récents. Néanmoins, cette maturité ne peut se concevoir sans l'extension des assistants à la preuve interactifs par des outils d'automatisation puissants.

L'objectif de ce travail est d'étendre le système Coq avec une tactique réflexive pour l'arithmétique réelle du premier ordre, qui repose sur une procédure de décision proche de l'état de l'art algorithmique des outils du calcul formel. J'ai en fait choisi de faire reposer la tactique sur une implémentation dans le système Coq de l'algorithme de calcul de CAD.

L'algorithme de CAD permet de résoudre le problème de décision car il calcule pour toute famille de polynômes une partition de l'espace en un nombre fini de cellules sur lesquelles chacun des polynômes de la famille de départ a un signe constant. L'algorithme procède par récurrence sur la dimension de l'espace, c'est à dire sur le nombre de variables des polynômes mis en jeu. Le cas de base des polynômes à une variable se traite avec des techniques d'isolation des racines réelles. Pour calculer cette partition en dimension supérieure, on effectue des projections successives en éliminant une par une les variables des polynômes de départ. À partir d'une famille $\mathcal{P}$ de polynômes à n variables, on calcule une nouvelle famille $Elim(\mathcal{P})$, et une partition de $\mathbb{R}^{n-1}$ qui est adaptée à $Elim(\mathcal{P})$. Cette partition est en fait la projection d'une partition adaptée à $\mathcal{P}$ selon la direction correspondant à la variable éliminée. Pour retrouver une partition adaptée à la famille $\mathcal{P}$, on étudie le cylindre au dessus de chaque cellule de la partition calculée récursivement pour $Elim(\mathcal{P})$. L'algorithme de CAD a une complexité meilleure que celle de ses prédécesseurs grâce à un procédé d'élimination efficace, qui utilise intensivement des calculs de *sous-résultants*. L'algorithme des sous-résultants est aussi une généralisation de l'algorithme d'Euclide qui permet de calculer efficacement des plus grands communs diviseurs de polynômes à coefficients dans un anneau, et en particulier de polynômes à plusieurs variables.

La conception de cette tactique vise à combler l'absence d'automatisation pour les problèmes d'arithmétique non-linéaires dans le système Coq. Mais sa réalisation représente un problème de formalisation intéressant en soi puisque la preuve de correction de l'algorithme de CAD est une preuve de taille conséquente (et ce déjà sur papier) et qui mêle des apports de théories variées, pour lesquelles il n'existe souvent pas de développement formel.

Cette thèse est organisée de la façon suivante.

Le premier chapitre propose une formalisation de l'arithmétique polynomiale à coefficients dans une structure d'anneau arbitraire en utilisant la forme dite de Horner creuse. Cette représentation possède deux traits caractéristiques. Tout d'abord elle permet une définition des polynômes à plusieurs variables, par un type dépendant de ce nombre de

variables. D'autre part, la formalisation réalisée permet des calculs relativement efficaces sur les polynômes.

Cette formalisation a été utilisée pour la programmation d'une tactique réflexive de décision des égalités sur les structures d'anneau et de semi-anneau. Cette tactique s'avère beaucoup plus efficace que son analogue documentée dans (Coq development team 2004), en particulier grâce à l'efficacité du calcul de la forme normale. La conception de cette tactique est un travail réalisé en collaboration avec B. Grégoire. La tactique est à présent intégrée à la version V8.1beta du système, grâce au travail de B. Barras, qui a ré-implémenté en Ocaml la partie frontale pour en faire un outil facile d'utilisation. Les preuves de la structure d'anneau des polynômes en forme de Horner ont été réalisées en collaboration avec L. Rideau.

Le second chapitre expose la formalisation de la preuve de correction d'un algorithme efficace pour le calcul de plus grand commun diviseur (pgcd) pour les polynômes à coefficients dans un anneau. Cet algorithme, qui repose sur le calcul des polynômes sous-résultants, permet en particulier de calculer des pgcd de polynômes à plusieurs variables. Il s'agit d'un algorithme standard du calcul formel, dont la preuve de correction est très technique, même sur papier. Ce travail constitue la première preuve formalisée de la correction de cet algorithme. Il a nécessité la formalisation en Coq de déterminants, qui repose sur une proposition de L. Théry. La preuve formelle de ce résultat n'est pas achevée, principalement du fait de difficultés techniques dues à l'implémentation actuelle des sétoïdes en Coq mais l'étape clef du théorème de correction est entièrement prouvée.

Le troisième chapitre ne contient pas de résultat original mais une tentative d'exposer les principes des algorithmes de décision pour l'arithmétique réelle. Il contient en particulier une description de l'algorithme de Décomposition Algébrique Cylindrique implémenté dans le système Coq, ainsi que des exemples illustrant les calculs qui constitue les pièces de ce puzzle du calcul formel. La preuve formelle des ingrédients du problème à une variable, en particulier l'utilisation des polynômes de Bernstein, sont un travail commun en cours avec F. Guilhot et Y. Bertot.

Le quatrième chapitre est consacré aux aspects algorithmiques de l'implémentation réalisée. Celle-ci est en particulier paramétrée par la représentation choisie par l'utilisateur pour les nombres rationnels et pour les nombres réels. D'autre part le style de programmation employé est fortement marqué par l'utilisation des types dépendants du système Coq.

Le cinquième et dernier chapitre propose une étude du contenu calculatoire d'un lemme standard d'analyse réelle, que l'on peut identifier à un principe d'induction ouverte. Une preuve intuitionniste de ce lemme nécessite l'utilisation d'un axiome, qui permette de raisonner sur des structures d'arbre à branchement dénombrable. Ce chapitre contient l'adaptation d'une preuve de T. Coquand à une structure de nombres réels munis d'ouverts énumératifs.

Chapitre 1

Arithmétique polynomiale certifiée

La première partie de ce chapitre décrit l'implémentation d'une bibliothèque d'arithmétique polynomiale dans le système Coq. On requiert seulement des coefficients qu'ils forment un anneau intègre décidable, ce qui permet donc de représenter les polynômes multivariés si les coefficients de base sont munis d'une telle structure.

La représentation choisie est délibérément spécialisée par rapport à une variable pour s'adapter à des problèmes présentant une structure récursive portant sur la dimension de l'espace. C'est en particulier le cas de l'algorithme de CAD que nous voulons implémenter. Les choix algorithmiques sont guidés par le double souci d'efficacité et de support à la preuve mécanisée, les preuves de corrections de toutes les opérations étant vérifiées dans le système.

La deuxième partie de ce chapitre est consacrée à la présentation d'une tactique réflexive prouvant les égalités dans une structure d'anneau qui repose sur une variante de cette représentation des polynômes. Cette tactique a été intégrée dans la version de développement courante de Coq. Ce sont les performances de cette tactique qui valideront l'efficacité de notre représentation.

1.1 Choix de la représentation

1.1.1 Une construction de l'anneau de polynômes

Cette section est consacrée à l'exposé d' une construction classique de l'anneau des polynômes à coefficients dans un anneau commutatif unitaire. Ce bref exposé est inspiré de (Chambert-Loir 2006).

Soit A un anneau commutatif et unitaire et d un entier naturel. Soit $\mathcal{P}_d$ le sous-ensemble de $A^{\mathbb{N}^d}$ formé des familles $(a_m)_{m\in\mathbb{N}^d}$ d'éléments de A, indexées par $\mathbb{N}^d$, dont presque tous les termes sont égaux à 0.

On vérifie sans difficulté que l'addition terme à terme munit $\mathcal{P}_d$ d'une structure de groupe abélien.

Soit $P = (p_m)_{m\in\mathbb{N}^d}$ et $Q = (q_m)_{m\in\mathbb{N}^d}$ des éléments de $\mathcal{P}_d$. Si $m \in \mathbb{N}^d$, il n'y a qu'un nombre fini de couples de multi-indices (m', m'') tels que $m = m' + m''$. On peut alors poser :

$$P \times Q := R = r_m = \sum_{m'+m''=m} p_{m'} q_{m''}.$$

La famille (r_m) est presque nulle, donc définit un élément de $\mathcal{P}_d$. On vérifie que cette loi $(P, Q) \mapsto R$ est associative et fait de $\mathcal{P}_d$ un anneau commutatif unitaire.

On peut également définir le produit scalaire d'un élement a de A par une suite presque nulle (p_m), défini par $(a * (p_m))$, qui propage sur la suite (p_m) le produit dans A par a. Ceci permet de munir $\mathcal{P}_d$ d'une structure de A-algèbre.

Soit $i \in \{1, \ldots, d\}$, notons X_i l'élément de $\mathcal{P}_d$ dont l'unique terme non nul est celui correspondant au multi-indice δ_i (qui vaut 1 en i et 0 ailleurs), et vaut 1. Si $P = (p_m)$, on a alors :

$$P = \sum_{m \in \mathbb{N}^d} p_m \prod_{i=1\ldots d} X_i^{m_i}$$

L'anneau $\mathcal{P}_d$ est appelé anneau des polynômes à d indéterminées $(X_i)_{i=1\ldots d}$ à coefficients dans A. On le note $A[X_1, \ldots X_d]$. Si $d = 1$, on le note $A[X]$ où X est l'indéterminée.

Pour $m \in \mathbb{N}^d$, l'expression $\prod_{i=1\ldots d} X_i^{m_i}$ est notée X^m et est appelée *monôme*, m_i est son degré en X^i et $\sum m_i$ est son degré total. Soit $P = \sum a_m X^m$, les monômes de P sont les X^m avec $a_m \neq 0$. Pour $i = 1 \ldots d$, on appelle degré de X_i de P et on note $\deg_{X_i}(P)$ le maximum des degrés en X_i des monômes de P. De même le degré total de P, noté $\deg(P)$, est le maximum des degrés totaux des monômes de P.

On répète ici à titre d'exemple la spécialisation de la construction ci-dessus au cas des polynômes à une seule variable.

Dans ce cas, on commence par considérer le sous-ensemble de $A^{\mathbb{N}}$ formé par les suites stationnaires à zéro d'éléments de A : $A[X] := \{(a_n)_{n\in\mathbb{N}} \mid \exists N, \forall n \geq N, a_n = 0\}$.

On définit ensuite X comme la suite dont le deuxième élément vaut 1_A et tous les autres 0_A. La structure de groupe de A se transporte naturellement sur $A[X]$ en définissant l'addition par l'opération terme à terme :

$$(a_n)_{n\in\mathbb{N}} + (b_n)_{n\in\mathbb{N}} := (a_n + b_n)_{n\in\mathbb{N}}$$

L'élément neutre est la suite identiquement nulle notée 0.
On munit ensuite $A[X]$ d'une loi produit définie par :

$$(a_n)_{n\in\mathbb{N}} * (b_n)_{n\in\mathbb{N}} := (a * b)_{n\in\mathbb{N}} \text{ où } (a * b)_n \sum_{k+l=n} a_k * b_l.$$

L'élément neutre est la suite nulle partout sauf en 1 où elle vaut 1. Ici, les vérifications que cette loi produit munit bien $A[X]$ d'une structure d'anneau commutatif unitaire ne posent aucune difficulté. On peut remarquer que ce produit agit comme un décalage à droite sur X : X^n est la suite partout nulle sauf en n où elle vaut 1. On obtient donc l'écriture unique d'un polynôme sous forme de somme coefficientée de monômes.

On remarque que comme $A[X]$ est ainsi doté d'une structure d'anneau commutatif unitaire, il peut à son tour servir d'ensemble de coefficients pour un anneau de polynômes. Par itération de cette construction, on obtient une seconde définition de l'anneau des polynômes à n variables $A[X_1, \ldots, X_n]$ comme $A[X_1] \ldots [X_n]$ [1].

[1] Pour se convaincre que ces définitions construisent des objets isomorphes, il faudrait définir la propriété universelle des algèbres de polynômes, voir par exemple (Lang 1995), ch. 5.

1.1.2 Représentation de Horner

Notre implémentation va encoder les polynômes sous forme de listes finies non vides, en utilisant la *représentation de Horner*. Ces listes présentent les coefficients de bas degré en tête. Une première possibilité pour définir ce type de données serait

```
Inductive Pol1(C:Set):Set:=
   |Pc : C → Pol1 C
   |PX : Pol1 C → C → Pol1 C.
```

Code 1.1: *Polynômes à une variable en forme de Horner*

L'ensemble `C` des coefficients est donné comme paramètre, `Pc` est le constructeur des polynômes constants et `PX` celui des polynômes de degré au moins 1. Le constructeur `PX` joue ici le même rôle que l'opérateur de *cons* des listes. La représentation d'un polynôme est unique modulo effacement des zéros de tête, et la taille de la forme normale d'un polynôme, c'est à dire sa représentation la plus compacte, est égale à son degré.

Exemple 1.1.2.1 *On considère des polynômes à coefficients dans* $\mathbb{Z}$.
- *Le polynôme constant* 1 *est représenté par* `(Pc 1)`
- *Le polynôme non constant* $X + 2$ *est représenté par* `(PX (Pc 1) 2)`
- *Le polynôme* X^4 *est représenté par* `(PX (PX (PX (PX (Pc 1) 0) 0) 0) 0)`

Le dernier exemple montre qu'une optimisation immédiate est possible pour compresser la représentation des polynômes creux (ceux qui présentent des sauts de degrés importants dans la suite de leurs monômes). On introduit donc un *indice de factorisation* au constructeur `PX`, sous la forme d'un argument entier strictement positif, qui permet de factoriser quand c'est possible des puissances de X. Cette représentation est appelée forme de Horner creuse.

```
Inductive Pol1(C:Set):Set:=
   |Pc : C → Pol1 C
   |PX : Pol1 C → positive → C → Pol1 C.
```

Code 1.2: *Polynômes à une variable en forme de Horner creuse*

L'ajout de ce nouvel argument augmente encore le nombre de représentations possibles pour un même polynôme, puisqu'il permet de représenter toutes les factorisations partielles des puissances de X. Néanmoins, on peut toujours définir une forme normale : la représentation sans zéro de tête et dans laquelle on a factorisé autant que possible. Cette représentation est la plus compacte possible et sa taille est le nombre de coefficients non nuls du polynôme.

Pour représenter cet indice de factorisation de façon à obtenir des calculs efficaces sur les puissances, nous avons utilisé la bibliothèque standard de `Coq` pour l'arithmétique en base deux. Dans cette bibliothèque, les nombres strictement positifs sont représentés par des suites de bits :
- `xH` représente 1
- `xO` x représente $2 \times x$
- `xI` x représente $(2 \times x) + 1$

Ceci se traduit en Coq par la définition inductive suivante :

```
Inductive positive : Set :=
  | xH : positive
  | xO : positive → positive
  | xI : positive → positive.
```

Code 1.3: *Représentation des entiers en base 2*

Par exemple, l'entier 6 est représenté par le terme `Coq (xO (xI xH))`. La bibliothèque standard fournit toutes les opérations arithmétiques sur ces entiers, ainsi que les opérations de comparaison.

Exemple 1.1.2.2 *On considère encore des polynômes à coefficients dans* $\mathbb{Z}$.

- *Le polynôme constant* 1 *est représenté par le terme* `(Pc 1)`
- *Le polynôme non constant* $X + 2$ *est représenté par le terme* `(PX (Pc 1) 1 2)`
- *Le polynôme* $X^{90}+3$ *est représenté en forme normale par le terme* `(PX (Pc 1) 90 3)` *mais il est aussi représenté par le terme* `(PX (PX (Pc 1) 89 0) 1 3)`

1.1.3 Opérations d'anneau

On suppose que l'ensemble des coefficients C est muni de deux constantes 0_C et 1_C, ainsi que des opérations d'anneau binaires $+$, $\times$ et $-$, et de l'opération unaire d'opposé $-$. On suppose de plus que l'égalité sur C est décidable, en particulier que l'on dispose d'un opérateur de test à zéro décidable.

La forme normale des polynômes étant aussi leur représentation la plus compacte, il est intéressant de maintenir cette représentation car la complexité des opérations (linéaire pour l'addition et la soustraction, quadratique pour le produit) dépendent de la taille des entrées.

On pourrait définir une fonction de normalisation qui serait appliquée systématiquement à la fin de chaque fonction qui rend un polynôme, pour garantir la préservation de la forme normale compacte, mais cette normalisation systématique est coûteuse. Nous avons opté pour une solution moins contraignante : on suppose que les polynômes fournis en argument de toutes les fonctions programmées sont en forme normale, et l'on garantit qu'alors, le résultat calculé par lesdites fonctions sera également en forme normale.

Programmer les opérations d'anneau sur cette représentation en forme de Horner creuse, en ne produisant que des résultats en forme normale, demande plus de travail du fait de l'introduction de l'indice de factorisation. En effet, pour maintenir la forme normale des polynômes, il faut tester des inégalités entre indices de factorisation.

En fait, à chaque fois qu'à l'intérieur d'une fonction une manipulation risque de briser la forme normale, soit en introduisant potentiellement un zéro en tête, soit en relâchant une factorisation de puissance de X, on effectue à cette place une normalisation *locale* adaptée. On définit à cet effet un *constructeur normalisant* de polynômes :

```
Definition mkPX P i c :=
  match P with
  | Pc p ⇒ if (czero_test p) then Pc c else PX P i c
  | PX P' i' c' ⇒
    if (czero_test c') then PX P' (i' + i) c else PX P i c
  end.
```

Code 1.4: *Constructeur normalisant pour les polynômes*

Le résultat de cette fonction est la forme normale de $P * X^i + c$, si le polynôme P en argument est en forme normale : si P est nul, alors il s'agit du polynôme constant c, si $P = P' * X^{i'}$, en forme normale, alors la forme normale attendue est $P' * X^{i'+i} + c$.

Par exemple si l'on veut calculer le résultat de la somme de `(PX P i p)` avec `(PX Q i q)`, deux polynômes en forme normale, alors comme `(P + Q)` peut être nul, on utilisera ce constructeur et on calculera `(mkPX (P + Q) i (p + q))`. Par contre toutes les fois que l'on sait par avance que l'invariant sera préservé, on pourra utiliser le constructeur `PX`, dont le coût est nul : par exemple la somme (en forme normale) de `(PX P i p)` (en forme normale) avec `(Pc c)` est bien `(PX P i (p +c))`.

Nous disposons maintenant des outils nécessaires pour programmer toutes les opérations d'anneau.

Opposé

On propage simplement sur la liste des coefficients l'opération d'opposé disponible sur l'ensemble des coefficients. Il n'est pas nécessaire ici d'utiliser de constructeur normalisant.

Multiplication

Pour la multiplication, on fournit d'abord la multiplication `Pol_mul_cst` par un polynôme constant `(Pc c)`, qui propage sur la liste des coefficients, le produit par `c`, puis on utilise `mkPX` pour construire le cas général. Comme on travaille dans un anneau intègre, on ne fait pas de test à zéro après un produit de deux éléments non nuls.

```
Fixpoint Pol_mul(P P':Pol) {struct P'} : Pol :=
 match P' with
 | Pc c' ⇒ Pol_mul_cst P c'
 | PX P' i' c' ⇒
    (mkPX (Pol_mul P P') i' c0) + (Pol_mul_cst P c')
 end.
```

Code 1.5: *Produit sur le polynômes*

Addition et Soustraction

L'addition et la soustraction sont également définies par récursion structurelle sur un des arguments mais un peu plus de travail est nécessaire. On détaille ici la construction de l'opérateur d'addition, celle de l'opérateur de soustraction se définit de façon équivalente, soit en suivant la même technique, soit comme addition de l'opposé.

On note ++ l'opération d'addition sur l'ensemble des coefficients `C`.

Tout d'abord, il est facile d'additionner un polynôme constant à un polynôme arbitraire : la somme se fait sur les éléments de tête de la liste (qui sont les coefficients constants). La fonction définie en figure 1.6 calcule la somme du polynôme `P` avec le polynôme constant `(Pc c)` :

```
Definition Pol_add_C P c :=
   match P with
    |Pc p → Pc (p ++ c)
    |PX P1 i p → PX P1 i (p ++ c)
   end.
```

Code 1.6: *Addition avec une constante*

Pour l'algorithme d'addition général, il faut prendre soin de maintenir la forme normale : supposons que l'on veut additionner $PX^i + p$ avec $QX^j + q$, deux polynômes en forme normale. Il y a trois cas :

- $i = j : (PX^i + p) + (QX^j + q) = (P + Q)X^i + (p + q)$
- $i > j : (PX^i + p) + (QX^j + q) = (PX^{i-j} + Q)X^j + (p + q)$
- $i < j : (PX^i + p) + (QX^j + q) = (P + QX^{j-i})X^i + (p + q)$

Dans chacun de ces trois cas, on rappelle la fonction d'addition sur un sous terme d'au moins l'un des deux arguments, cette définition est donc bien formée.

L'opération ci-dessus utilise une soustraction pour pouvoir comparer les indices, qui sont eux strictement positifs. On utilise donc ici la bibliothèque d'entiers *relatifs* construite au dessus du type `positive` décrit dans la section 1.1.2, et qui définit les entiers relatifs par cas sur leur signe :

```
Inductive Z : Set :=
  | Z0 : Z
  | Zpos : positive → Z
  | Zneg : positive → Z.
```

Code 1.7: *Entiers en base deux*

L'opération `ZPminus`, qui prend en argument deux entiers strictement positifs, représentes par des termes de type `positive` décrit dans la section calcule leur différence dans $\mathbb{Z}$ et renvoie un résultat de type `Z`.

Les définitions par point fixe acceptées automatiquement par le système `Coq` doivent être des récursions structurelles sur l'un des arguments, ce qui garantit facilement la terminaison de la fonction ainsi définie. Si une telle définition n'est pas possible, l'utilisateur doit d'abord définir l'ordre qui doit assurer la terminaison et prouver sa bonne fondation. La fonction sera ensuite définie grâce à une preuve d'accessibilité des arguments, mais ce type de définition est assez technique à mettre en place.

Il est possible ici de contourner cette définition par récursion bien fondée et de définir les fonctions d'addition et de soustraction par deux points fixes structurels.

Ici le rôle des arguments est symétrique, supposons donc que la récursion portera dans le cas non trivial où les deux arguments sont non constants, sur le second argument, $QX^j + q$. Les deux premiers cas ci-dessus font effectivement des appels légitimes à la fonctions d'addition à droite de Q, mais pas le dernier cas. En effet, on y fait appel à la fonction `fun P:Pol → P + Q X^k`. Cette dernière est effectivement définissable par

induction sur P à l'aide d'appels à **fun** P:Pol $\mapsto P + QX^k$, mais un deuxième point fixe imbriqué sera nécessaire.

On va ainsi d'abord programmer le calcul de P + Q $\times X^k$ par récursion sur P, connaissant Q et **fun** P:Pol $\mapsto P + QX^k$:

```
Fixpoint Pol_addX(Paux:Pol→Pol)(Q:Pol)
    (i':positive)(P:Pol) {struct P} : Pol :=
    match P with
      | Pc c ⇒ PX Q i' c
      | PX P i c ⇒
        match ZPminus i i' with
         | Zpos k ⇒ mkPX (Paux (PX P k c0)) i' c
         | Z0 ⇒ mkPX (Paux P) i c
         | Zneg k ⇒ mkPX (Pol_addX k P) i c
        end
    end.
```

Code 1.8: *Fonction auxiliaire d'addition sur les polynômes*

On peut définir l'opérateur d'addition, par récursion structurelle sur le deuxième argument, en utilisant Pol_addX lorsqu'il faudra détruire le premier.

```
Fixpoint Pol_add (P P': Pol) {struct P'} : Pol :=
  match P' with
    | Pc c' ⇒ Pol_addC P c'
    | PX P' i' c' ⇒
      match P with
        | Pc c ⇒ PX P' i' (c ++ c')
        | PX P i c ⇒
          match ZPminus i i' with
            | Zpos k ⇒
               mkPX (Pol_add (PX P k c0) P') i' (c ++ c')
            | Z0 ⇒
               mkPX (Pol_add P P') i (c ++ c')
            | Zneg k ⇒
               mkPX
               (Pol_addX (fun x ⇒ Pol_add x P') P' k P) i (c ++ c')
          end
      end
  end.
```

Code 1.9: *Opérateur d'addition sur les polynômes*

On peut de façon équivalente définir la soustraction comme l'addition de l'opposé ou bien en utilisant la même construction que pour l'addition.

Cet artifice qui consiste à écrire deux points fixes imbriqués pour contourner les contraintes de la récursion structurelle en Coq semble peu naturel. Dans ce cas simple, il suffirait de pouvoir former la récursion sur la somme des constructeurs des arguments pour pouvoir assurer la borne formation du point fixe. Le travail présenté dans (Balaa et Bertot 2002) puis dans (Barthe, Forest, Pichardie, et Rusu 2006), améliore grandement l'aisance de définition d'une telle fonction dans le système.

La mesure qui décroît à chaque appel récursif de la fonction `Pol_add` est le nombre total de constructeurs `PX` dans les arguments de l'opération. On définit donc une fonction :

```
Definition total_number_of_PX(u:Pol*Pol): nat.
```

qui totalise le nombre de constructeurs du couple de polynômes en arguments. Il est ensuite possible de définir l'opération d'addition, en précisant que l'on utilise la mesure `total_number_of_PX` pour mesurer la décroissance de l'argument à chaque appel récursif.

```
Function Pol_add(c:Pol*Pol){measure total_number_of_PX }:Pol :=
   match c with
     |(P1, P2) ⇒
       match P1 with
         |Pc c1 ⇒ Pol_addC P2 c1
         |PX P1' i p1 ⇒
           match P2 with
             |Pc c2 ⇒ Pol_addC P1 c2
             |PX P2' j p2 ⇒
               match (ZPminus i j) with
                 |Z0 ⇒
                    mkPX (Pol_add (P1', P2')) i (p1 ++ p2)
                 |Zpos k' ⇒
                    mkPX (Pol_add ((PX P1' k' c0), P2')) j (p1 ++ p2)
                 |Zneg k' ⇒
                    mkPX (Pol_add (P1', (PX P2' k' c0))) i (p1 ++ p2)
               end
           end
       end
   end.
```

Code 1.10: *Code de l'addition en utilisant la construction* Function

À ce stade de la programmation, le système génère les obligations de preuves qui doivent être démontrées : on doit prouver que la mesure des arguments décroît à chaque appel récursif, ce qui assure la terminaison en prouvant la bonne fondation de l'ordre.

Ainsi le système Coq génère les buts représentés ci-dessous :

```
3 subgoals

     ∀ (c : Pol * Pol) (P1 P2 : Pol),
   c = (P1, P2) →
   ∀ (P1' : Pol) (i : positive) (p1 : Coef),
   P1 = PX P1' i p1 →
   ∀ (P2' : Pol) (j : positive) (p2 : Coef),
   P2 = PX P2' j p2 →
   ZPminus i j = 0 →
      total_number_of_PX (P1', P2') < total_number_of_PX c
```

```
subgoal 2 is:
 ∀ (c : Pol * Pol) (P1 P2 : Pol),
 c = (P1, P2) →
 ∀ (P1' : Pol) (i : positive) (p1 : Coef),
 P1 = PX P1' i p1 →
 ∀ (P2' : Pol) (j : positive) (p2 : Coef),
 P2 = PX P2' j p2 →
 ∀ k' : positive,
 ZPminus i j = Zpos k' →
  total_number_of_PX (PX P1' k' c0, P2') < total_number_of_PX c


subgoal 3 is:
 ∀ (c : Pol * Pol) (P1 P2 : Pol),
 c = (P1, P2) →
 ∀ (P1' : Pol) (i : positive) (p1 : Coef),
 P1 = PX P1' i p1 →
 ∀ (P2' : Pol) (j : positive) (p2 : Coef),
 P2 = PX P2' j p2 →
 ∀ k' : positive,
 ZPminus i j = Zneg k' →
 total_number_of_PX (P1', PX P2' k' c0) < total_number_of_PX c
```

Code 1.11: *Énoncés des conditions de décroissance de la mesure sur les appels récursifs de l'addition*

Ces énoncés ont une preuve triviale : c'est à dire qu'ils sont prouvés directement par la tactique **`auto with arith`** qui essaie de prouver le but à partir d'une base de lemmes élémentaires.

Cet outil est récent, il n'était pas encore développé au moment où j'ai programmé la bibliothèque, c'est pourquoi je ne l'ai pas testé autrement que sur cet exemple jouet. Le gain en confort dans cette définition est très important : il permet d'écrire d'utiliser sans effort la récursion bien fondée et de se passer des points fixes imbriqués.

Malheureusement, le code de la fonction d'addition produit par la définition ci-dessus, n'est pas du tout efficace pour être réduit dans le système. Cette approche sera donc plus adaptée aux développements destinés à être extraits que pour produire des fonctions exécutées par la machine de réduction de Coq elle-même. Pour obtenir du code qui s'exécute efficacement en Coq, tout en profitant de cette automatisation, il faut probablement définir deux versions du code, l'une pour les preuves et l'autres pour le calcul, ainsi qu'une preuve de leur égalité extensionnelle.

1.1.4 Égalité et preuves

Il s'agit maintenant de transmettre à la structure de polynômes `(Pol1 C)` les relations et propriétés disponibles sur la structure de coefficients `C`. L'ensemble `C` est muni d'une relation d'égalité décidable, notée $=_C$. On suppose aussi que constantes et opérations d'anneau de C sont compatibles avec la relation d'égalité $=_C$ et vérifient les axiomes d'anneau. Pour ne pas surcharger les notations, on omet les indices lorsqu'il n'y a pas d'ambiguïté.

- $\forall x, 0 + x =_C x$
- $\forall xy, x + y =_C y + x$
- $\forall xyz, x + (y + z) =_C (x + y) + z$
- $\forall x, 1 * x =_C x$
- $\forall xy, x * y =_C y * x$
- $\forall xyz, x * (y * z) =_C (x * y) * z$
- $\forall xyz, (x + y) * z =_C (x * z) + (y * z)$
- $\forall xy, x - y =_C x + -y$
- $\forall x, x + (-x) =_C 0$

FIG. 1.1: *Axiomes de la structure d'anneau*

Il s'agit de prouver les propriétés d'anneau pour les opérations et constantes dont nous avons donné l'implémentation ci-dessus.

Le choix de la relation d'égalité (paramétrée par une relation d'égalité sur C) dont on veut munir l'ensemble des polynômes est déterminant pour la nature des preuves à venir.

Le fait que plusieurs termes de type `Pol` correspondent à un même objet mathématique se traduit par l'existence d'une relation d'équivalence dont les classes contiennent exactement les différentes représentations d'un même polynôme. La structure d'anneau que l'on va construire s'appuie sur le quotient de l'ensemble des termes représentant les polynômes par cette relation.

On distingue trois phases dans la construction formelle de la structure d'anneau sur les polynômes.

- La définition de la relation et les preuves qu'il s'agit d'une relation d'équivalence ;
- Les preuves de compatibilité des opérations avec la relation d'équivalence choisie ;
- Les preuves des axiomes de la structure d'anneau.

Il existe plusieurs choix pour la définition de cette relation d'équivalence.

Égalité dans le quotient

La solution que nous avons adoptée est de définir un prédicat d'égalité qui décrit toutes les variations que les différentes représentations peuvent introduire en décrivant sous quelles conditions deux polynômes, éventuellement formés par des constructeurs différents, peuvent représenter le même objet mathématique. Ce prédicat se définit en `Coq` comme ci-dessous. Le polynôme nul est noté `P0` et la constante nulle `c0`.

```
Inductive Pol_Eq:Pol → Pol → Prop :=
(* Deux polynomes constants sont egaux si les constantes sont
egales au sens des coefficients*)
  |Eq_Pc_Pc : ∀ p q : Coef, (ceq p q) ⇒ (Pol_Eq (Pc p) (Pc q))

(* Un polynome constant p et un polynome non constant PX^i + q sont
egaux si p et q sont des constantes egales et P est nul *)
  |Eq_Pc_PX :
    ∀ p q : Coef, ∀ P : Pol,
    (ceq p q) ⇒ (Pol_Eq P P0) ⇒ ∀ i, (Pol_Eq (Pc p) (PX P i q))
  |Eq_PX_Pc :
    ∀ p q: Coef, ∀ P : Pol,
```

```
    (ceq p q) ⇒ (Pol_Eq P P0) ⇒ ∀ i, (Pol_Eq (PX P i p) (Pc q))

(* Si p et q sont des constantes egales et P et Q deux polynomes
egaux, alors PX^i+p et QX^i+q sont egaux*)
  |Eq_PX_PX :
    ∀ p q : Coef, ∀ P Q : Pol,
    (ceq p q) ⇒ (Pol_Eq P Q) ⇒ ∀ i, (Pol_Eq (PX P i p) (PX Q i q))

(* Si Q est egal à PX^i et si p et q sont deux
 constantes egales, PX^i+p et QX^i+q sont egaux *)
  |Eq_PXi_PXij:
    ∀ p q : Coef, ∀ P Q : Pol, ∀ i j,
    (ceq p q) ⇒ (Pol_Eq Q (PX P i c0)) ⇒
     (Pol_Eq  (PX Q  j   q)(PX P (i+j)  p))

(* Le cas symetrique du precedent *)
  |Eq_PXij_PXi:
    ∀ p q : Coef, ∀ P Q : Pol, ∀ i j,
    (ceq p q) ⇒ (Pol_Eq Q (PX P i c0)) ⇒
     (Pol_Eq (PX P (i+j)  p) (PX Q j q)).
```

Code 1.12: *Prédicat d'égalité sur les polynômes*

Les propriétés d'anneau sont ensuite prouvées directement, c'est à dire qu'elle portent sur l'implémentation des opérations. Un prédicat inductif comme celui-ci permet de générer à l'induction tous les cas pertinents d'une hypothèse d'égalité, et seulement ceux-là.

Une autre possibilité pour raisonner sur cette relation serait de la définir par l'égalité pseudo-syntaxique des formes normales de deux polynômes. Par égalité pseudo-syntaxique, on entend ici l'égalité structurelle modulo la relation d'égalité sur les coefficients, qui ne sera pas en général une égalité de Leibniz. Dans le cas des polynômes en forme de Horner, cette dernière solution est trop coûteuse. Elle demande de raisonner avec deux niveaux de setoïdes (égalité pseudo-syntaxique et égalité des formes normales) et l'on doit souvent utiliser des tactiques de nettoyage car les inductions génèrent de nombreux cas non pertinents.

Nous avons achevé la preuve formelle complète qui munit l'ensemble des polynômes de Horner d'une structure d'anneau pour la relation d'équivalence du code 1.12 et les opération décrites dans la section 1.1.3. Nénamoins cette preuve, bien qu'aboutie, n'est pas totalement satisfaisante. En particulier, l'utilisation d'un indice de factorisation introduit beaucoup de technicité dans les preuves, du fait des conditions de comparaison sur ces puissances.

Une approche alternative ?

Le caractère fastidieux des preuves mentionné ci-dessus doit être nuancé :

- Prouver qu'une représentation efficace d'un objet mathématique correspond bien au modèle théorique comprend une certaine part incontournable de preuves techniques. Les preuves que la relation fixée est une relation d'équivalence, et celles que les opérations sont bien compatibles peuvent être intrinsèquement difficile si des optimisations fines ont été introduites.

– Par contre, il est remarquable que dans la stratégie décrite ci-dessus, dans toutes les preuves faisant intervenir ces opérations, comme c'est le cas pour les propriétés d'anneau, on retrouve des parties techniques similaires, dues aux optimisations.

Il serait intéressant de pouvoir en quelque sorte *abstraire* la preuve de ces optimisations, en les prouvant *une fois pour toute*. Il faut pour cela montrer que la représentation en forme de Horner, munie des opérations programmées est isomorphe à une autre représentation des polynômes, plus adaptée aux preuves.

Nous proposons un nouveau schéma pour le développement formel, dont le cheminement est en fait général, et dépasse le cadre de l'exmple des polynômes :

– formaliser (une deuxième fois) une représentation des polynômes en Coq, dans un style choisi pour que les preuves formelles soient aisées sur cette nouvelle représentation ;
– monter un isomorphisme entre les polynômes en forme de Horner et la nouvelle représentation. C'est à ce moment que l'on montre que les optimisations introduites dans la représentation pour faciliter les calculs ne changent pas l'objet mathématique ;
– dériver une relation d'égalité sur les polynômes de Horner à partir de cet isomorphisme ;
– montrer que chaque opération est compatible avec cette relation d'équivalence : on montre ainsi que les optimisations introduites dans chacune de ces opérations ne modifient pas leur sens mathématique.
– retrouver les propriétés d'anneau, en héritant de leur analogue sur la représentation choisie pour les preuves.

Il existe une construction mathématique des polynômes dans laquelle les propriétés d'anneau sont données gratuitement, par définition, il s'agit de l'algèbre libre des polynômes (Lang 1995). Dans cette structure, c'est la définition même de la relation d'égalité qui donne la structure d'algèbre puisqu'on quotiente le domaine par les axiomes requis.

L'algèbre libre *Pol* des polynômes à une variable X et à coefficients dans un anneau A est caractérisée par la propriété universelle suivante.

Proposition 1.1.1 (Propriété universelle) *Soit A un anneau et X un symbole de variable. On appelle Pol l'algèbre libre des polynômes à une variable X et à coefficients dans un anneau A. On note i l'injection de $\{X\}$ dans Pol. Pour toute A-algèbre B, et pour toute fonction $f : \{X\} \mapsto B$, il existe une fonction $lift : Pol \mapsto B$ telle que :*

$$lift \circ i = f$$

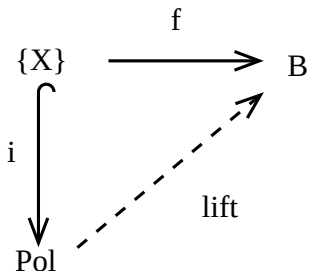


FIG. 1.2: *Propriété universelle de l'algèbre libre des polynômes*

La fonction $lift$ de la propriété ci-dessus correpond en fait à l'évaluation des polynômes en la valeur déterminée par la fonction f.

La définition de l'algèbre libre des polynômes sur un ensemble de variables donné, ainsi que la preuve de la propriété universelle ont été formalisé par L. Pottier[2].

Pour utiliser ce développement, il faut montrer que la structure des polynômes de Horner peut être identifiée à l'algèbre Pol par un isomorphisme. Il s'agit donc de construire une fonction $\phi : Horner \mapsto Pol$ où $Horner$ est en fait représenté en Coq par `(Pol1 C)` pour un anneau de coefficients `C`. Pour que ϕ puisse être un isomorphisme entre les deux structures, on définit une égalité $=_h$ sur $Horner$ qui identifie les fibres de ϕ :

$$\forall x, y \in Horner, \quad \phi(x) =_{Pol} \phi(y) \Leftrightarrow x =_h y$$

Lorsqu'on montre que cette relation est une relation d'équivalence, on montre que l'on identifie bien les différentes représentations d'un même polynôme, de la même façon que le prédicat `Pol_Eq`.

Puis il faut prouver que l'application ϕ est surjective, c'est à dire qu'on dispose bien d'une représentation en forme de Horner pour *tous* les polynômes :

$$(*) \quad \forall y \in Pol, \exists x \in Horner \text{ tq } \phi(x) =_{Pol} y$$

Pour chaque opération d'anneau sur les polynômes en forme de Horner, il faut montrer le lemme de compatibilité avec ϕ, ce qui donne en fait le lemme de compatibilité avec la relation $=_h$. Par exemple pour une opération binaire $\Box$, ce lemme s'écrira :

$$(**) \quad \forall x, y \in Horner, \phi(x \Box y) =_{Pol} \phi(x) \Box \phi(y)$$

La preuve de ce lemme condense en fait la preuve des optimisations apportées par la version $Horner$ des opérations.

Enfin, les preuves des propriétés d'anneau sont héritées gratuitement de la structure de l'algèbre libre grâce à $(*)$ et $(**)$.

Cette approche n'efface pas la difficulté intrinsèque des preuves avec les versions optimisées des polynômes, mais elle permet de n'être confronté qu'une seule fois aux parties techniques de ces preuves (analyses par cas sur les partiés d'exposant, tests à zéro,...) qui ne relèvent pas de la structure mathématique.

Le modèle de l'agèbre libre des polynômes peut être remplacé par n'importe quelle structure jugée plus judicieus, qui est alors prise comme formalisation de référence pour les polynômes (listes de coefficients, flux de coefficients, ...). Le choix présenté dans cette section offre l'avantage d'être la structure canonique. Ainsi d'une part on prouve effectivement que les polynômes de Horner sont bien *la* structure de polynômes, mais aussi on a pas besoin de travailler pour obtenir la structure d'algèbre du modèle de référence.

[2]dans une extension de la contribution Algebra (Pottier 1999) qui contient une hiérarchie algébrique qui va des ensembles aux nombres complexes.

1.2 Une tactique réflexive pour les égalités dans un anneau

Nous présentons ici une application directe d'une telle implémentation d'arithmétique polynomiale élémentaire. Il s'agit de la programmation d'une tactique réflexive pour automatiser les preuves d'égalités dans les structures d'anneau. Ce travail est détaillé dans (Grégoire et Mahboubi 2005).

Pour l'utilisateur d'un système de calcul formel, l'une des fonctionnalités du système les plus intéressantes est peut-être celle de simplifications des expressions symboliques, utilisant en particulier des identités algébriques. Dans un système de calcul formel comme Maple, une telle fonction des simplification utilise une grande bibliothèque d'identités algébriques (ou autres), qui sont combinées pour établir le résultat désiré.

Fournir un tel outil à un assistant à la preuve demande de programmer une procédure de décision, puisqu'il s'agit dans ce contexte de fournir également une preuve de l'identité établie. Nombre de ces identités qu'un utilisateur d'assistant à la preuve aura à établir sont en fait des égalités dans des structures d'anneau $(\mathbb{R}, \mathbb{Z}, \dots)$ ou de semi-anneau $(\mathbb{N}, \dots)$: identités remarquables comme la célèbre $(a+b)^2 = a^2 + 2ab + b^2$, réécriture modulo associativité/commutativité,...

Il est possible de décider ces égalités d'anneau (ou de semi-anneau), il semble donc indispensable de fournir un outil d'automatisation de ces preuves, ça l'établissement de l'identité :

$$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$$

nécessiterait l'application successive de pas moins d'une trentaine d'étapes de réécriture des axiomes de l'anneau sous-jacent.

Un tel outil, appelé `ring`, existe dans la distribution standard de Coq. Développé par S. Boutin (Boutin 1997), ce fut l'exemple historique de développement d'une tactique *réflexive* dans le système Coq. Néanmoins, cet outil souffre d'un manque d'efficacité important, en particulier pour les structures non calculatoires. Prouver à l'aide de `ring` l'identité $100 * 100 = 10000$ est immédiat si les constantes sont interprétées comme des entiers dans $\mathbb{Z}$, mais cela prendra une centaine de secondes si les entiers sont vus comme des nombres réels (dans Coq, $\mathbb{R}$ est une structure axiomatique). Ce mauvais comportement sur les constantes affecte bien sûr du même coup l'efficacité de la méthode sur les identités algébriques de haut degré.

Enfin, le code de l'actuelle procédure `ring` est présenté sous la forme de huit implémentations distinctes, dépendant du type de structure : axiomatique ou calculatoire, anneau ou semi-anneau, avec une égalité de Leibniz ou une structure de setoïde. Nous proposons une implémentation modulaire de notre procédure de décision, grâce à un code unique, qui est instanciable grâce la donnée de trois paramètres pour s'adapter à une quelconque de ces combinaisons.

Dans toute la suite, nous appellerons `newring` la nouvelle procédure que nous décrivons dans cette section.

Jusqu'à la section 1.2.4, nous ne parlerons que de structures d'anneaux, mais tout ce qui suit s'applique également aux structures de semi-anneaux. La section 1.2.4 expliquera comment traiter ces deux structures d'une façon unifiée.

1.2.1 Réflexion

Dans le système Coq, les étapes de réécritures dans une preuve sont explicites : chacune de ces étapes construit un prédicat qui a la taille du but courant au moment où la réécriture est effectuée et de ce fait la taille du terme de preuve dépend lourdement du nombre de ces étapes de réécritures. La technique dite de réflexion, introduite par (Allen, Constable, Howe, et Aitken 1990), tire parti de la machine de réduction de l'assistant à la preuve pour réduire la taille du terme de preuve calculé et par conséquent accélérer sa vérification. Elle peut en particulier être utilisée comme une alternative aux preuves par réécriture dans les cas des théories décidables. Elle repose sur la remarque suivante :

- Soit $P : A \mapsto Prop$ un prédicat sur un ensemble A ;
- Supposons que nous sommes capables d'écrire dans le système une procédure de semi-décision f, telle que f soit calculable et f rend le résultat `true` sur l'entrée x dès que $P(x)$ est valide. Ainsi on a :

```
f_correct: ∀ x, f(x)=true → P(x)
```

si l'on veut prouver $P(y)$ pour une valeur y particulière et si nous savons que $f(y)$ se *réduit* à `true`, alors il suffit d'applique le théorème `f_correct` à y et à une preuve que $true = true$. Rappelons maintenant que la règle de conversion :

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s \qquad T \equiv U}{\Gamma \vdash t : U}$$

autorise à changer implicitement le type d'un terme par un autre, pourvu que ces deux types soient équivalents (modulo réduction). Notre preuve de `true=true`, qui est le terme Coq `(refl_equal true)` est aussi une preuve que `f(y)=true` puisque dès lors que `f(y)` se réduit à `true`, `f(y)=true` et `true=true` sont convertibles. La preuve de `P(y)` que nous avons construite est ainsi :

```
f_correct y (refl_equal true)
```

La taille d'une telle preuve dépend seulement de l'argument y et du nombre d'étapes implicites de réduction : les étapes explicites de réécritures ont été remplacées par des étapes implicites de réduction. La taille de la preuve du théorème de correction de f peut être importante, elle ne sera faite qu'une seule fois, sera partagée par toutes les instanciations et, une fois établie, ne sera plus jamais vérifiée. Bien sûr, l'efficacité de cette technique dépend fortement de l'efficacité de la procédure de décision f elle-même ainsi que de l'efficacité du système à réduire le terme $f(y)$. Pour ce dernier point, l'introduction récente d'une machine abstraite et d'un compilateur au système Coq encourage le développement de ce type d'approche pour le développement de tactiques de décision.

Remarquons que pour assurer la correction de ce procédé, la complétude de f n'est pas nécessaire, d'où son qualificatif de procédure de *semi*-décision. Néanmoins dans notre cas, nous serons capable de fournir facilement une preuve papier de complétude, qui assure que la tactique "prouve tout ce qui est prouvable".

La tactique `newring` opère sur une structure d'anneau A, incluant un type pour le domaine de ses éléments, également nommé A, deux constantes 0 et 1, trois opérations binaires $\times$, $-$ et $+$, et une opération unaire d'opposé $-$, ainsi que les propriétés (ou axiomes) usuelles définissant une structure d'anneau commutatif unitaire. Il s'agit de prouver l'égalité de deux termes t_1 et t_2 de type A, modulo la structure d'anneau.

Travailler par réflexion signifie que nous voulons construire une procédure de semi-décision f, prenant t_1 et t_2 en argument, et qui ne rend la valeur `true` que si t_1 et t_2 sont égaux modulo réécriture associative-commutative dans la structure d'anneau.

Une façon naturelle d'effectuer une comparaison entre deux termes serait d'utiliser un filtrage pour les séparer ou les identifier. Néanmoins, Coq ne permet pas un tel filtrage sur des termes quelconques, mais seulement sur des termes ayant un type inductif. C'est pourquoi les termes de A vont être *réfléchis* dans un type inductif approprié $PolExpr$, qui décrit la syntaxe des termes de A. Cette étape est aussi appelée *réification*. Un terme de A est réfléchi par une méta-fonction $\mathcal{T}$ vers une expression polynomiale dans $PolExpr$ en

- interprétant chaque constante de l'anneau comme une expression polynomiale constante (par exemple 0, 1, mais il peut y en avoir d'autre...)
- interprétant chaque opération d'anneau comme une opération sur les expressions polynomiales
- cachant chaque sous-terme qui n'est ni une constante de l'anneau, ni l'application d'une opération d'anneau à d'autres sous-termes derrière une variable étiquetée et en conservant la liste d'association correspondante.

La fonction $\mathcal{T}$ peut être considérée comme un oracle, et nous expliquerons en 1.2.5 comment il est possible de la programmer en utilisant le *méta-langage* Ltac. Une fois construites les deux expressions polynomiales e_1 et e_2, de type $PolExpr$, correspondant respectivement à t_1 et t_2, il s'agit de tester l'égalité des formes normales de e_1 et e_2 et de prouver que leur égalité implique celle de t_1 et t_2. Pour ce faire nous devrons assurer la correction du diagramme suivant :

$$\begin{array}{lccccl} e_1 = e_2 & PolExpr & \xrightarrow{norm} & Pol & & norm(e_1) = norm(e_2) \\ & \mathcal{T}\uparrow \quad \downarrow \varphi_{PE} & & \downarrow \varphi_P & & \\ t_1 = t_2 & A & \iff & A & & \varphi_P(norm(e_1)) = \varphi_P(norm(e_2)) \end{array}$$

en prouvant le lemme suivant de correction vis-à-vis de l'évaluation :

$$\forall e \in PolExpr,\ \varphi_{PE}(e) = \varphi_P(norm(e))$$

Les fonctions φ_{PE} (resp. φ_P) sont des fonctions d'évaluation. Elles évaluent les expressions polynomiales (resp. les polynômes normalisés Pol) en les éléments de A, en ré-interprétant chaque constante polynomiale en une constante de l' anneau A, chaque variable en le terme qu'elle cachait et chaque opération sur les polynômes en l'opération d'anneau correspondante.

Ces fonctions peuvent être définies facilement à l'intérieur de la théorie par filtrage sur les types inductifs réfléchis $PolExpr$ et Pol. Pour assurer la complétude de notre tactique, il faut prouver la "méta"-propriété suivante :

$$\forall a \in A.\ \varphi_{PE}(\mathcal{T}(a)) = a.$$

Encore une fois, il n'est pas nécessaire de prouver formellement cet énoncé dans le système Coq, puisque ce résultat n'affecte pas la correction de la méthode mais seulement sa complétude.

Le type inductif *PolExpr* est adapté à la réification puisque c'est une formalisation de la syntaxe des termes de A. Le type *Pol* quant à lui représente le type des formes normales pour les expressions polynomiales, et n'est pas nécessairement identique à *PolExpr*. Il doit par contre être adapté à un calcul efficace de forme normale, la fonction *norm* faisant le lien être ces deux types aux contraintes différentes.

Finalement, pour prouver l'égalité de deux termes t_1 et t_2, on commence par calculer e_1 et e_2 les expressions polynomiales réifiées correspondantes, grâce à $\mathcal{T}$, puis on calcule et compare leurs formes normales. Si elles sont égales, alors les termes t_1 et t_2 sont égaux par le lemme de correction et la transitivité de l'égalité :

$$t_1 = \varphi_{PE}(\mathcal{T}(t_1)) = \varphi_P(norm(\mathcal{T}(t_1))) = \varphi_P(norm(\mathcal{T}(t_2))) = \varphi_{PE}(\mathcal{T}(t_2)) = t_2$$

1.2.2 Creuser encore la représentation des polynômes

Après cette présentation générale du schéma de réflexion qui sous-tend notre tactique, nous allons décrire plus précisément la forme normale choisie. Il s'agit d'un des facteurs important de l'efficacité de la procédure. Il s'agit donc de donner plus de détails sur le type *Pol* ci-dessus.

Dans la section précédente 1.1, nous avons proposé une représentation des polynômes univariés en forme de Horner. Il est aisé d'étendre cette représentation au cas général des polynômes multivariés en construisant par un point fixe l'itération de la construction.

Nous avions proposé une optimisation (autoriser les factorisations de l'indéterminée) pour combler les creux dans la suite des puissances d'un polynôme et ne représenter que la suite des coefficients non nuls. Ici un deuxième type de creux apparaît dans la représentation pour les polynômes multivariés : les sauts de variables. Par exemple le polynôme constant 1 sera représenté (en forme normale) par le terme `(Pc 1)` s'il est vu comme un élément de $\mathbb{Z}[X]$, mais sera représenté (toujours en forme normale) par le terme `(Pc (Pc (Pc (Pc 1))))` s'il est vu comme un élément de $\mathbb{Z}[X, Y, Z, T]$. Pour remédier à cet inconvénient, nous abandonnons ici l'idée de définir les polynômes récursivement par rapport au nombres de variables, à partir des polynômes à une variable. On définit en un seul coup l'ensemble des polynômes à coefficients de base dans C, avec nombre arbitraire de variables par :

```
Inductive Pol (C:Set) : Set :=
 | Pc : C → Pol C
 | Pinj : positive → Pol C → Pol C
 | PX : Pol C → positive → Pol C → Pol C.
```

Code 1.13: *Représentation optimisée des polynômes multivariés*

La sémantique des constructeurs est la suivante :

- `(Pc c)` représente le polynôme constant $c \in C[X_1, \ldots X_n]$ pour n quelconque ;
- si Q est un élément de $C[X_1, \ldots, X_{n-j}]$ et `Q` sa représentation, alors `Pinj j Q` représente le polynôme $Q.X^0_{n-j+1} * \ldots * X^0_n$, c'est à dire Q "poussé" de l'espace $C[X_1, \ldots, X_{n-j}]$ vers l' espace des polynômes à n variables $C[X_1, \ldots, X_n]$. L'entier j est appelé indice d'injection ;
- enfin, `PX P i Q` représente $P*X^i_n+Q$ où $P \in C[X_1, \ldots, X_n]$ et $Q \in C[X_1, \ldots, X_{n-1}]$ est constant en X_n la variable courante.

Notre dernière optimisation augmente encore le nombre de représentations pour un polynôme. Là encore, il est facile de définir une forme normale sur cette représentation, qui sera la plus compacte de toute. Nous adoptons toujours la même stratégie : au lieu de construire une fonction de normalisation qui construit un représentant canonique pour chaque polynôme et qui serait appliqué avant tout test d'égalité, nous choisissons de ne manipuler que des représentants canoniques, c'est à dire ceux qui vérifient :

- Le coefficient de tête n'est jamais nul.
- L'indice de factorisation est toujours le plus grand possible (ie. vu la condition précédente, il n'y a jamais de coefficient nul dans la représentation du polynôme, sauf éventuellement le dernier).
- L'indice d'injection est toujours le plus grand possible.

Pour préserver cette forme normale, comme dans la section précédente 1.1, les opérations sur les polynômes devront construire des formes normales. Or on est toujours dans le cas où il est facile de construire la forme normale de `PX P i Q` en détruisant localement `P`, lorsque `P` et `Q` sont déjà en forme normale :

- si `P = (Pc 0)` alors le représentant canonique cherché est celui de `(Pinj 1 Q)` ;
- si `P = PX P'  i (Pc 0)` alors le représentant canonique est `(PX P'(i+i') Q)` ;
- sinon `(PX P i Q)` est le représentant canonique recherché.

Il nous faudra cette fois deux *constructeurs normalisants*, `mk_PX` et `mk_Pinj`, pour reprendre la terminologie utilisée en 1.1, pour effectuer dans le corps des opérations les normalisations locales nécessaires à la préservation de la forme normale.

Les algorithmes permettant le calcul des opérations d'anneau sur le type `Pol` ne sont pas très différents de ceux exposés en section 1.1, à ceci près qu'il faut tenir compte du nouveau constructeur `Pinj`.

Pour chaque opération d'anneau il nous faudra cette fois prouver, non plus les propriétés axiomatisant une structure d'anneau, mais un lemme de correction vis-à-vis de l'évaluation dans `C`. Par exemple pour l'addition sur les polynômes, le lemme s'énonce :

```
Lemma Padd_correct: ∀ P Q l,
     phiP l (Padd P Q) == (phiP l P) + (phiP l Q).
```

où `==` est l'égalité (possiblement une égalité de setoïde) dont est munie la structure initiale A d'anneau (ou de semi-anneau) et où `+` est l'addition sur A.

Construire la fonction de normalisation des expressions polynomiales vers leur forme normale de Horner consiste à envoyer les constructeurs de constantes vers des polynômes constants, les variables vers des monômes et les constructeurs d'opération vers les fonctions sur les formes de Horner. La forme normale de l'expression polynomiale est obtenue en évaluant le terme obtenu.

Nous devons également prouver la correction de la normalisation vis-à-vis de l'évaluation dans A, c'est-à-dire prouver formellement le lemme suivant :

```
Lemma norm_correct : ∀ l e, phiPE l e == phiP l (norm e).
```

A ce point du développement, nous sommes en mesure d'énoncer (et de prouver) le théorème de correction de notre schéma de réflexion :

```
Lemma f_correct : ∀ l e1 e2,
  Peq (norm e1) (norm e2) = true → phiPE l e1 == phiPE l e2.
```

où `Peq` représente une fonction définie dans le développement et qui teste l'égalité syntaxique entre deux formes de Horner.

Nous avons décrit ici la forme normale des polynômes que nous manipulons, et les algorithmes d'opérations polynomiales. Néanmoins, le domaine sur lequel la fonction de normalisation effectue ses calculs est l'ensemble de coefficients C des polynômes et expressions polynomiales. La section suivante montre que le choix de C est crucial pour l'efficacité de la procédure et qu'il peut même enrichir le domaine d'application de la tactique elle-même car cet ensemble C doit capturer la meilleur composante calculatoire de l'anneau.

1.2.3 Calculs sur un ensemble de coefficients paramétré

L'efficacité de la fonction de normalisation que nous avons décrite ci-dessus est conditionnée par le comportement calculatoire de l'ensemble de coefficients choisis. Par exemple, la normalisation du polynôme $x+(-x)$ conduira à $((1+(-1)).x)$ qui se *réduit* à $0.x$. Ainsi, l'ensemble C doit être judicieusement choisi comme un ensemble sur lequel la réduction effectue les calculs attendus, et ce le plus efficacement possible. Dans le système `Coq`, ces ensembles sont représentés par des types inductifs, et les opérations sont des programmes fonctionnels.

Dans le système `Coq`, `Z` est une implémentation de $\mathbb{Z}$ où les entiers sont représentés par des listes de chiffres binaires. Dans les cas où `Z` est la structure sous-jacente à l'égalité que l'on veut prouver, `Z` lui-même est un bon candidat pour le choix de l'ensemble des coefficients.

Par contre, si l'anneau sous-jacent est `R`, la représentation axiomatique des réels distribuée dans la bibliothèque standard de `Coq`, `R` lui-même ne sera pas un choix judicieux comme ensemble de coefficients. En effet dans `R`, $1 + (-1)$ est bien égal à 0, en utilisant les axiomes d'anneau, mais ne se *réduit* pas à 0 : la soustraction, de même que les autres opérations sur `R`, ne sont que des symboles non évaluables. C'est pourquoi $x+(-x)$ ne serait pas réduit vers $0.x$ par la fonction de normalisation. Néanmoins, l'injection naturelle de $\mathbb{Z}$ dans $\mathbb{R}$ nous permettra dans ce cas de choisir `Z` comme ensemble de coefficients et de résoudre ce problème. Quelle que soit la structure d'anneau A avec laquelle nous travaillons, le morphisme canonique de $\mathbb{Z}$ dans A nous permettra d'utiliser encore `Z` comme ensemble de coefficients. Ce type `Z` ne serait-il pas alors le candidat universel pour toute structure d'anneau ?

En fait `Z` ne sera pas toujours le meilleur choix possible. Si le contenu calculatoire de l'anneau est plus puissant que les seuls axiomes de la structure d'anneau, choisir un ensemble de coefficient éventuellement différent de l'anneau de départ va nous permettre de prouver plus d'égalités que celles que l'on doit aux seules réécritures des axiomes de structure d'anneau. Considérons le cas où l'anneau dans lequel nous travaillons est `bool`. Dans cette structure, l'identité $x + x = 0$ est valide quel que soit x élément de l'anneau. Néanmoins, cette identité n'est pas prouvable par réécriture des axiomes d'anneau (sinon elle serait vraie dans tout anneau...). Cette fois, le bon choix pour l'ensemble des coefficients est `bool` lui-même : en effet le terme $x + x$ sera réfléchi vers l'expression polynomiale $X + X$ (à coefficients dans `bool`). La forme normale $(1 + 1).X$ de cette expression est *réduite* vers $0.X$, grâce aux calculs sur `bool`. C'est pourquoi notre choix est de paramétrer la tactique par cet ensemble de coefficients et de laisser l'utilisateur

faire le choix le plus approprié.

Toutefois, un type inductif devra satisfaire certaines conditions pour être un ensemble de coefficients admissible. Ces conditions sont celles qui permettent d'assurer la correction de la fonction de normalisation. Formellement, un ensemble C sera admissible comme ensemble de coefficients s'il est équipé des constantes et opérations d'anneau, ainsi que d'une relation d'égalité décidable $=_C$. Cette dernière condition est nécessaire à l'implémentation des constructeurs `mk_Pinj` et `mk_PX` (car ils exigent des tests à zéro) et elle permet ensuite de muni l'ensemble des polynômes en forme de Horner d'une égalité décidable. On demande surtout l'existence d'une fonction d'évaluation de C vers A, qui envoie les constantes de C vers des éléments de A, et qui doit être compatible avec les opérations respectives de C (au départ) et A (à l'arrivée). On résume cet ensemble de contraintes en disant que l'ensemble C est admissible comme ensemble de coefficients s'il existe un morphisme entre C et A (même si C n'est pas nécessairement un anneau). Ce morphisme évalue les constantes et opérations de C en leurs analogues dans A et la relation $=_C$ doit vérifier : si $x =_C y$ est vrai, alors les évaluations de x et y seront égales dans A, au sens de l'égalité de A.

Une fois choisi cet ensemble de coefficients C et prouvées ces dernières spécifications de morphismes, nous définissons de manière générique les opérations sur les polynômes à coefficients dans C comme décrit en section 1.2.2. Nous étendons ensuite le morphisme de C vers A en deux fonctions d'évaluation $\varphi_P E$ et φ_P, évaluant respectivement les expressions polynomiales et les polynômes en forme de Horner en des éléments de A. Nous obtenons ainsi une preuve du diagramme proposé en 1.2.1, Pol et $PolExpr$ étant remplacés par leur version paramétrée $Pol(C)$ et $PolExpr(C)$.

Nous avons implémenté le cas du morphisme identité, pour lequel l'ensemble des coefficients choisi est l'anneau lui-même. L'utilisateur pourra éventuellement se servir de la tactique qui en résulte, même si ce choix ne permet pas de prouver beaucoup d'identités (comme c'est le cas dans `R`). Nous avons aussi implémenté le morphisme générique qui permet de choisir `Z` comme ensemble de coefficients, pour un anneau arbitraire. Comme mentionné ci-dessus, ceci peut-être utilisé comme un choix par défaut satisfaisant, même si cela peut ne pas être le meilleur (ce ne sera pas le cas par exemple dans le cas d'un anneau calculatoire $\mathbb{Z}/n\mathbb{Z}$ comme `bool`).

Pour tirer le bénéfice maximal de cette méthode, l'utilisateur doit faire le choix de l'ensemble de coefficients le plus approprié. Si la structure d'anneau dans laquelle il travaille est définie de façon axiomatique, sans contenu calculatoire, comme dans le cas de `R`, le choix de `Z` est toujours le plus judicieux. Dans le cas où l'anneau présente un contenu calculatoire, comme `Z` ou `bool`, prendre l'anneau lui-même comme ensemble de coefficient peut s'avérer un bien meilleur choix. Cependant `Z` peut rester le plus adapté à un calcul aussi efficace que possible, si les opérations de l'anneau, sont, certes évaluables mais moins efficaces que celle de la représentation des entiers binaires `Z` (c'est le cas par exemple du semi-anneau des entiers de Péano)

1.2.4 Unification des structures d'anneaux et de semi-anneaux

Un semi-anneau est une structure algébrique munie de deux opérations binaires $+$ et $*$, de deux constantes 0 et 1 et définie par les mêmes axiomes que ceux d'une structure d'anneau, à ceci près que les axiomes qui définissaient le comportement de l'opposé et de

la soustraction sont remplacés par une unique propriété $0 * x = x$. Par exemple l'ensemble des entiers naturels $\mathbb{N}$ présente une structure de semi-anneau.

Ces deux types de structures demeurent très semblables et l'on attendrait d'un outil travaillant sur les structures d'anneaux qu'il puisse s'adapter aux structures de semi-anneaux sans duplication de code. Pour ceci nous allons travailler avec une structure intermédiaire que nous appelleront *pseudo-anneau*. L'idée principale de cette définition est de compléter une structure de semi-anneau par l'ajout d'un opérateur unaire, qui est moralement l'opérateur d'opposé d'une structure d'anneau et que nous appellerons *pseudo-opposé*. Cet opérateur sera instancié par une fonction triviale lorsqu'il s'agira de munir un semi-anneau d'une structure de pseudo-anneau.

En fait la remarque fondamentale est la suivante : pour les structures d'anneaux, dans la preuve de correction de la fonction de normalisation, l'axiome définissant l'opposé comme un *inverse*, qui affirme que $\forall x, x + (-x) = 0$, n'est jamais utilisé en temps que tel, mais on a seulement besoin des propriétés qui décrivent le comportement de l'opposé lorsqu'il est combiné avec d'autres opérations. C'est pourquoi un pseudo-anneau sera défini par les propriétés suivantes :

- $\forall x,\ 0 + x = x$
- $\forall x\ y,\ x + y = y + x$
- $\forall x\ y\ z,\ x + (y + z) = (x + y) + z$
- $\forall x,\ 1 * x = x$
- $\forall x\ y,\ x * y = y * x$
- $\forall x\ y\ z,\ x * (y * z) = (x * y) * z$
- $\forall x\ y\ z,\ (x + y) * z = x * z + y * z$
- $\forall x, 0 * x = 0$ (ici nous avons tous les axiomes d'un semi-anneau)
- $\forall x\ y,\ -(x * y) = -x * y$ (combinaison du pseudo-opposé avec le produit)
- $\forall x\ y,\ -(x + y) = -x + -y$ (combinaison du pseudo-opposé avec l'addition)
- $\forall x\ y,\ x - y = x + -y$ (définition d'une pseudo-soustraction)

FIG. 1.3: *Axiomes de la structure de pseudo-anneau*

Il est immédiat de vérifier que tout anneau est un pseudo-anneau. Par contre, les axiomes d'un semi-anneau ne permettent pas de retrouver l'axiome supprimé de la liste des propriétés fondamentales d'un anneau, à savoir $x + (-x) = 0$. Néanmoins, notre tactique prouvera cette identité dès lors que dans l'ensemble de coefficients, $1 + (-1)$ se *réduit* vers 0. Or cette condition est assurée lorsque l'on a établi l'existence d'un morphisme de l'ensemble des coefficients vers l'anneau (par correction de l'évaluation des constantes et opérations). Quant aux structures de semi-anneau, elles peuvent facilement être étendue en structures de pseudo-anneau si l'on prend comme opérateur d'opposé l'identité, et comme soustraction l'opérateur d'addition déjà disponible dans la structure.

Finalement, il suffit donc de programmer la tactique en supposant que la structure de départ est un pseudo-anneau. On construit ensuite de façon générique les preuves que l'on peut transformer un anneau ou semi-anneau arbitraire en un pseudo-anneau. Si l'on travaille sur un semi-anneau il suffit de rajouter les pseudo-opérateurs nécessaires et si l'on travaille sur un anneau, on doit juste prouver les trois dernières propriétés, qui ne sont pas syntaxiquement des axiomes de la structure d'anneau mais dont les preuves sont triviales. Comme indiqué ci-dessus, c'est le calcul qui permettra à cette tactique de

prouver les identités d'anneau qui ne sont pas vraies dans un semi-anneau.

1.2.5 Programmer la réification et la tactique avec Ltac

Il nous reste à expliciter les étapes de construction de la tactique qui ne se font pas dans la théorie du système, mais au niveau du *métalangage*. Il y a essentiellement deux telles étapes. La première est la programmation de l' "oracle" $\mathcal{T}$ introduit en section 1.2.1, qui permet d'associer à un terme de l'anneau (sous-terme d'un membre de l'égalité à prouver) une expression polynomiale. La deuxième enfin est l'assemblage final de la tactique générique et son instanciation par les paramètres nécessaires (coefficients, type de structure et type d'égalité). Nous présenterons ici les facilités qu'offre le méta-langage Ltac (Delahaye 2000; Bertot et Casteran 2004), introduit dans le système Coq depuis sa version 7.0) pour définir ces deux programmes. Cette solution offre l'avantage d'être rapide et légère puisque tout est programmable à toplevel, et sans rentrer dans les détails de la syntaxe abstraite des termes Coq.

Il s'agit donc d'abord de programmer l'étape de réification $\mathcal{T}$. En effet, notre tactique se propose de prouver des buts de la forme $t_1 == t_2$, en appliquant le lemme `f_correct`. Pour ceci, nous devons produire une liste de valeur l et deux expressions polynomiales e_1 et e_2 telles que l'évaluation de e_1 (resp. e_2) en l soit convertible avec t_1 (resp. t_2). Par exemple, dans le cas de l'égalité suivante :

$$3 * \sin(x) * x = x * (\sin(x) + 2 * \sin(x)) + 0 * y$$

l sera $[\sin(x);\ x;\ y]$, e_1 sera $3 * X_1 * X_2$ et e_2 sera $X_2 * (X_1 + 2 * X_1) + 0 * X_3$.

Le langage Ltac nous offre de grandes facilités de filtrage sur les termes qui vont nous permettre de définir une telle opération de façon naturelle et très semblable à l'implémentation de la tactique `field`, décrite dans (Delahaye et Mayero 2001). Dans tout ce qui suit, nous appelons *tactique* tout programme écrit dans le langage Ltac (mais non nécessairement destiné à être utilisé dans un contexte de preuve).

D'abord, nous allons construire une fonction `FV` qui calcule la liste l des termes à abstraire. Ces termes sont ceux qui ne peuvent pas être décrits par la syntaxe d'anneau (comme $\sin(x)$ dans l'exemple ci-dessus), cette fonction est donc définissable par filtrage sur chacun des membres de l'égalité. Puis, la tactique `mkPolexpr` va calculer les deux expressions polynomiales e_1 et e_2 en utilisant l pour savoir derrière quelle variable cacher une sous-expression à abstraire donnée.

```
Ltac mkPolexpr Cst add mul sub opp t l :=
 let rec mkP t :=
  match t with
  | (add ?t1 ?t2) ⇒
    let e1 := mkP t1 in
    let e2 := mkP t2 in constr:(PEadd e1 e2)
  | (mul ?t1 ?t2) ⇒ ...
  | (sub ?t1 ?t2) ⇒ ...
  | (opp ?t1) ⇒ ...
```

```
  | _ ⇒
    match Cst t with
    | tt ⇒ let p := Find_at t l in constr:(PEX p)
    | ?c ⇒ constr:(PEc c)
    end
  end
 in mkP t.
```

Code 1.14: *Réification en* Ltac

La tactique `mkPolexpr` prend comme argument le terme `t` auquel on veut associer une expression polynomiale, la liste `l` des termes à abstraire, les noms des opérations d'anneau et *une tactique* `Cst`. Elle va filtrer le symbole de tête de `t` :

- si ce symbole de tête est une des opérations d'anneau, alors elle construit récursivement l'expression polynomiale associée
- si ce symbole de tête n'est pas un telle opération, alors soit `t` est une constante, soit `t` doit être abstrait en une variable. Cette distinction est faite par la tactique `Cst` :
  - Si `Cst` retourne la valeur `false`, alors l'indice de la variable appropriée est sa position dans la liste `t`.
  - Sinon, `t` est envoyé vers la constante appropriée.

De fait l'ensemble des constantes, et donc la tactique `Cst` dépend de l'anneau sous-jacent, ainsi que de l'ensemble de coefficients choisis. La tactique `Cst` implémente la réciproque du morphisme défini en 1.2.3.

Si A l'anneau sous-jacent est défini de façon axiomatique, alors nous pouvons utiliser une tactique naïve qui ne filtre que les constantes d'anneau `r0` et `r1` et les envoient sur leurs analogues dans `Z` :

```
Ltac genCstZ rO rI t :=
  match t with
   | rO ⇒ constr:0
   | rI ⇒ constr:1
   | _ ⇒ constr:tt
   end.
```

Code 1.15: *Morphisme générique pour des coefficients dans* `Z`

Cette tactique permet l'implémentation de la stratégie appliquée par défaut aux structures axiomatiques (voir 1.2.3). Par contre si A est `Z` lui-même, l'ensemble des coefficients sera encore `Z` mais nous pouvons filtrer plus de constantes, en fait toutes celles qui sont formées uniquement de constructeurs de `Z` :

```
Ltac ZCst t :=
 match (is_ZCst t) with
 | true ⇒ constr:t
 | false ⇒ constr:false
 end.
```

Code 1.16: *Morphisme amélioré pour l'anneau* `Z`

Ici `is_ZCst` est une tactique qui reconnaît les termes formés avec les seuls constructeurs du type inductif `Z`.

Cette méthode se généralise sans peine aux structures de semi-anneau, pour laquelle `N`, implémentation en `Coq` des entiers en base deux, joue le rôle précédemment tenu par `Z`.

Il s'agit maintenant d'assembler toutes ces pièces pour construire la tactique à proprement parler. Cette tactique est générique : lorsqu'elle est instanciée par les paramètres appropriés, elle fournit un outil qui prouve les égalités dans la structure d'anneau ou de semi-anneau décrite par les paramètres.

Nous allons utiliser ici les possibilités qu'offre `Ltac` pour définir des fonctions d'ordre supérieur. Dans un souci de clarté de l'exposé nous ne présentons d'abord que la version de la tactique qui prouve l'égalité du but courant ou bien échoue.

```
Ltac Make_ring_tac add mul sub opp req Cst_tac :=
  match goal with
  | [ |- req ?r1 ?r2 ] ⇒
    let fv := FV Cst_tac add mul sub opp (add r1 r2) nil in
    let e1 := mkPolexpr Cst_tac add mul sub opp r1 fv in
    let e2 := mkPolexpr Cst_tac add mul sub opp r2 fv in
    apply (f_correct fv e1 e2); compute; exact (refl_equal true)
  | _ ⇒ fail "not equality"
  end.
```

La tactique prend en paramètre les opérations de la structure (ici un anneau) et une tactique `Cst_tac` qui comme expliqué ci-dessus discrimine les constantes de la structure et les envoie vers les constantes de la structure de coefficients C choisie.

Elle vérifie d'abord que le but courant est bien une égalité (de l'anneau) entre deux termes `r1` et `r2`. Puis, si c'est le cas, elle calcule la liste globale `fv` des termes à abstraire dans les deux membres, en calculant celle du terme `r1 + r2`. Puis elle calcule les deux expressions polynomiales `e1` et `e2` associées respectivement à `r1` et `r2`. Enfin la tactique applique le lemme `f_correct`.

A ce point de l'exécution, le but courant est maintenant la vérification de l'hypothèse de `f_correct`, à savoir que `(norm e1) == (norm e2)` est syntaxiquement égal à `true` (où le test d'égalité `==` est celui de l'ensemble des polynômes de Horner à coefficients dans C).

Si `r1` et `r2` sont effectivement égaux modulo la structure d'anneau, ce dernier but est convertible à `true = true`. Il est donc possible de terminer la preuve en fournissant au système un terme ayant le type `true = true`, qui est dans `Coq (ref_equal true)`. La tactique **`exact`** vérifie que le terme qu'on lui fournit en argument a bien un type convertible à celui qui est attendu pour résoudre le but courant.

La tactique **`exact`** effectue donc les réductions nécessaires pour vérifier que `(norm e1) == (norm e2)` est convertible à `true`. Cependant la stratégie de réduction utilisée par **`exact`** n'est pas la plus efficace, c'est pourquoi afin d'accélérer les calculs, on effectue un appel préalable à la tactique **`compute`** pour réduire efficacement le terme.

1.2.6 Performances

La tactique `newring` propose deux améliorations orthogonales aux choix effectués dans le développements de `ring` (Boutin 1997). La première est de remplacer la forme

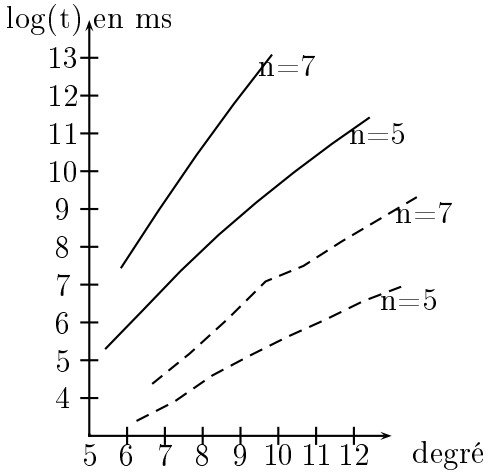


FIG. 1.4: *Temps de calcul de la preuve que* $(x_1 + \ldots + x_n)^d$ *est égal à sa forme normale*

normale totalement développée de polynômes représentés comme somme ordonnée de monômes coefficientés par des formes de Horner. La seconde est d'utiliser `Z` comme ensemble de coefficients lorsqu'on travaille avec des structures axiomatiques comme `R`. Dans cette section, nous analysons les gains en efficacité apportés par chacun de ces traits.

Tout d'abord, lorsqu'on compare les performances de `newring` et `ring` sur des expressions dans `Z`, on ne tire aucun bénéfice de la seconde amélioration puisque les deux tactiques travaillent sur leur représentation des polynômes à coefficients dans `Z`. Sur ce genre de problèmes, on mesure donc uniquement l'influence du changement de forme normale.

La figure 1.4 décrit les temps de normalisation pour le polynôme $(x_1 + \cdots + x_n)^d$ vu comme un polynôme à coefficients dans `Z`. Les traits pointillés représentent les temps de normalisation par `newring` et les traits pleins ceux de la normalisation par `ring`.

Sur ce diagramme on observe que le gain pour $n = d = 5$ est un facteur 6, et un facteur 500 pour $n = 7$ et $d = 9$, ceci grâce à la compacité de la forme normale de Horner. De plus la tactique `ring` n'arrive plus à calculer la forme normale de cette expression pour $n = 8$ et $d = 9$ et lorsque $n = 12$, le calcul n'est plus possible à partir de $d = 6$. Quant à `newring`, elle arrive encore à calculer la forme normale pour $d = 11$ et $n = 12$.

Comparer un outil comme `newring` avec un analogue dans le monde du calcul formel est intéressant mais la comparaison a ses limites. Bien sur les systèmes de calcul formel travaillent sur des véritables entiers machines alors que nous ne manipulons ici qu'un encodage en `Coq` des entiers en base 2. Mais surtout les ambitions et, partant, les stratégies adoptées divergent fortement. Dans un système de calcul formel, il s'agit de donner rapidement des réponses sur des entrées qui peuvent être de très grande taille. Ainsi dans `Maple V`, la fonction `expand` travaille à l'aide d'une bibliothèque d'identités pré-calculées, qu'elle essaie de recombiner pour simplifier l'expression donnée par l'utilisateur. Sur des expressions symboliques construites à l'aide d'opérations d'addition, produit et soustraction, `expand` essaie de développer en simplifiant si possible (les produits par 0, par 1, ...), visiblement sans avoir d'autre stratégie très optimisée, mis à part cette reconnaissance des motifs présents en mémoire. En effet, lorsque les identités algébriques sont inutilisables, le système s'avère très inefficace, voire incapable d'effectuer le calcul.

Néanmoins ce problème arrive plus rapidement que l'on pourrait s'y attendre : nous avons testé `expand` contre `newring` sur l'expression :

$$\begin{array}{c}(y + x_2 + \ldots + x_{n-1} + x_n) * \\ (x_1 + y + \ldots + x_{n-1} + x_n) * \\ \vdots \\ (x_1 + x_2 + \ldots + y + x_n) * \\ (x_1 + x_2 + \ldots + x_{n-1} + y)\end{array}$$

Pour $n = 8$ la tactique `newring` est quatre fois plus lente que la stratégie `expand` de Maple (0.4s pour `newring`, 0.12s pour Maple). Mais Maple échoue au développement de cette formule dès que $n = 9$ (`Error, (in expand/bigprod) object too large`), alors que pour cet exemple, `newring` termine en 1.7s.

1.3 Polynômes dans les systèmes de preuves formelles

A notre connaissance, il existe au moins cinq développements dans le système Coq qui comprennent une telle bibliothèque d'arithmétique polynomiale.

Le premier est inclus dans la contribution Algebra[3] au système Coq.Dans ce développement, on part d'un ensemble de variables fini V pour définir les monômes comme éléments du monoïde libre abélien sur V, puis les polynômes à coefficients dans l'anneau A et à variables V seront les éléments du monoïde libre abélien sur les monômes et sur A.

La seconde[4] est issue du projet Constructive Coq Repository at Nijmegen (C-CoRN 2002; Cruz-Filipe, Geuvers, et Wiedijk 2004). Elle consitute une partie de la hiérarchie d'algèbre constructive (Geuvers, Pollack, Wiedijk, et Zwanenburg 2002) éloborée dans le but de donner une preuve constructive en Coq du théorème fondamental de l'algèbre.

Les polynômes à une variable et à coefficients dans un anneau de coefficients `CR` sont représentés en forme de Horner. Un polynôme est :

- soit un polynôme nul,
- soit un polynôme de la forme $PX + c$, récursivement défini par un polynôme P et une constante c, élément de l'anneau des coefficients.

En Coq, cette définition correspond à la déclaration de la structure de données inductive :

```
Inductive cpoly : Type :=
  | cpoly_zero : cpoly
  | cpoly_linear : CR → cpoly → cpoly.
```

Le fait d'introduire un constructeur pour le polynôme nul permet d'alléger les preuves en remplaçant les tests à zéros décidables par des analyses par cas sur le terme du type inductif.

On définit une relation d'équivalence qui identifie les polynômes structurellement égaux modulo l'égalité sétoïde définie sur l'anneau des coefficients `CR`. Les polynômes

[3] http ://coq.inria.fr/contribs-eng.html, L. Pottier, 1999.

[4] http ://c-corn.cs.ru.nl/documentation/ CoRN.algebra.CPolynomials.html

sont munis des structures successives de setoïde, semi-groupe, groupe, monoïde, anneau, de la hiérarchie abstraite.

Ce développement est fortement marqué par l'objectif de développer des mathématiques constructives.

Les similitudes dans les choix des trois derniers développements sont assez frappantes, même si aucune ne semble avoir bénéficié de l'existence de ses prédécesseurs.

Le plus ancien que nous avons recensé est celui qui sous-tend la tactique `ring` distribuée actuellement dans la version standard de Coq. On peut également citer GbCoq, une certification de l'algorithme de Buchberger par L. Théry et L. Pottier décrite dans (Théry 2001). Enfin, le plus récent est une bibliothèque[5] sur la réécriture et la terminaison, développée par un groupe de contributeurs européens et hébergée au Loria.

Ces trois développements ont en effet adopté le même choix initial pour fonder leur représentation : un polynôme, qu'il soit univarié ou multivarié, est une somme coefficientée de monômes. C'est à dire qu'on représente un polynôme par une liste de couples coefficient-monôme, un monôme étant représenté par la liste ordonnée de ses degrés non nuls relatifs à chacune des indéterminées.

Ainsi dans les commentaires du code de la tactique `ring`, on lit :

"Définition des polynômes abstraits en forme normale :

- Une liste de variables est un produit ordonné d'une ou plusieurs variables : X, $X * Y * Z$ sont des listes de variables.
- Un monôme est une constante, une liste de variables ou bien un produit d'une constante par une liste de variables : $2 * X * Y$, $X * Y * Z$ sont des monômes, $2 * 3$, $X * 3 * Y$, $4 * X * 3$ n'en sont pas.
- Une somme canonique est soit un monôme, soit une somme ordonnée de monômes (selon un ordre précisé ultérieurement).
- Un polynôme en forme normale est soit une constante, soit une somme canonique, soit la somme d'une constante et d'une somme canonique".

Dans cette bibliothèque, on dispose d'une implémentation des opérations d'anneau et des preuves des propriétés de correction de ces opérations vis-à-vis de l'évaluation dans l'anneau des coefficients. On y définit également une forme normale, qui est la forme totalement développée, dans laquelle les monômes sont ordonnés lexicographiquement par rapport à un ordre fixé sur les variables.

La bibliothèque développée par L. Théry fait le même choix théorique mais propose une implémentation différente, plus abstraite. En effet, on paramètre la représentation par une notion de *terme*, chaque terme étant composé d'un coefficient et d'un monôme. On doit fournir des primitives sur ces termes : opérations d'anneau, ordre, égalité, et puis les polynômes seront définis comme des listes ordonnées de termes non nuls.

Contrairement à la bibliothèque précédente, dans laquelle une fonction de normalisation donne les représentations canoniques, ici les preuves des propriétés de bonne formation (ordre et non nullité des éléments) sont données comme arguments des constructeurs de polynômes.

Cette dernière bibliothèque implémente plus que l'arithmétique d'anneaux : on dispose également d'une division euclidienne (pour un anneau intègre de coefficients), d'un opérateur de plus petit commun multiple et bien sûr de calcul de base de Gröbner. Le

[5] A Coq Library on Rewriting and termination, http ://color.loria.fr/, 2005

développement est destiné à être exécuté par Ocaml, après extraction de la partie calculatoire et en instanciant les paramètres par des fonctions dans Ocaml.

La bibliothèque Color est au contraire très spécialisée dans les besoins des preuves sur les systèmes de réécriture. Elle fournit une représentation dans laquelle l'ensemble des coefficients est fixé comme étant $\mathbb{Z}$, et les polynômes sont encore une fois des listes de couples coefficient et monômes. Les monômes sont des vecteurs d'entiers de taille fixée.

```
Notation monom := (vector nat).
Definition poly n := (list (Z * monom n)).
```

Les opérations d'anneau, ainsi que l'évaluation, la composition sont fournies mais les preuves ne concernent que le comportement des opérations par rapport à l'évaluation et les questions de monotonies utiles aux interprétations polynomiales des ordres de réécriture.

L'avantage de ce choix commun aux trois exemples cités est qu'il correspond à une vision globale du polynôme, et de ses propriétés combinatoires comme la symétrie ou l'homogénéité. En revanche, les informations relatives à une variable particulière (degré, factorisations par rapport à une variable) ont un coût. Les performances comparées des deux implémentations de la tactique d'égalité dans les anneaux montrent également que le calcul de la forme normale d'un polynôme dans cette représentation n'est pas optimal.

Dans le système Nuprl (Constable, Allen, Bromley, Cleaveland, Cremer, Harper, Howe, Knoblock, Mendler, Panangaden, Sasaki, et Smith 1986), P. Jackson a développée une formalisation d'une structure abstraite de polynômes, en suivant comme L. Pottier la construction canonique de (Lang 1995).

La thèse de C. Ballarin (Ballarin 1999) décrit l'implémentation d'une bibliothèque pour les polynômes en Isabelle (Nipkow, Paulson, et Wenzel 2002), comme fonctions presque nulle des entiers vers l'anneau des coefficients.

Les travaux de J. Harrison en HOL-Light (Harrison 2002; Harrison 1996) utilisent extensivement des objets polynomiaux, qui sont une des premières théories (Harrison 1997) formalisées dans le système.

Les déveoppements ultérieurs réalisés en HOL-Light mêlent calcul scientifique et preuve formelle, que ce soit en important les réponses d'oracles externes (Harrison et Théry 1998; Harrison 2005) ou en programmant des tactiques autarciques pour le système (McLaughlin et Harrison 2005).

Dans ce système, les polynômes sont définis comme des fonctions d'évaluation : un polynôme est une fonction d'évaluation appliquée à une liste de coefficients et à un élément de l'ensemble des coefficients qui est le point où on veut l'évaluer.

la liste vide représente le polynôme nul.

```
let poly = new_recursive_definition list_RECURSION
  `(poly [] x = &0) ∧
   (poly (CONS h t) x = h + x * poly t x)`;;
```

Code 1.17: *Définition des polynômes en* HOL-Light

La distribution standard du système comprend de nombreux résultats (ordre des racines, décomposition en partie sans carré, valeurs intermédiaires dans les nombres réels...), dont les preuves ont une remarquable concision.

1.4 Conclusion

Nous proposons une bibliothèque d'arithmétique polynomiale basée sur la représentation de Horner creuse et destiné à calculer efficacement à l'intérieur du système Coq. Cette bibliothèque se décline en deux versions.

La première est une représentation des polynômes à une variable, paramétrée par un ensemble de coefficients. On peut itérer cette construction pour construire un type des polynômes à plusieurs variables, dépendant du nombre variables. Pour cette représentation, les propriétés de la structure d'anneau ont été formellement prouvées. La technique employée s'est toutefois révélée coûteuse en temps de travail et nous avons proposé une méthode alternative pour munir d'une structure mathématique des représentations formelles orientées vers le calcul.

La deuxième version des polynômes en forme de Horner creuse supprime la dépendance du type en le nombre de variables. Cette représentation a été utilisée pour la conception d'une tactique d'automatisation des preuves d'égalités dans les structures d'anneau. Pour les besoins de cette tactique, nous n'avons pas explicitement construit la structure d'anneau formée par les polynômes mais nous avons donné les preuves formelles des propriétés des opérations vis à vis du morphisme d'évaluation.

La rénovation que nous proposons de la tactique réflexive d'automatisation des preuves d'égalités dans les structures d'anneau introduit deux améliorations orthogonales qui améliore son efficacité. Le premier est l'utilisation de la forme de Horner creuse pour les polynômes, le second est d'exploiter au mieux les possibilités de calcul dans la phase de réification. On peut néanmoins isoler les effets du changement de forme normale, et donc valider l'efficacité de ce choix de représentation.

Une tactique efficace pour la décision des égalités dans les structures d'anneau est un outil fondamental pour un système de calcul formel. Le fait de pouvoir manipuler des expressions de taille importante même dans des structures axiomatiques perment d'envisager de façon réaliste des développements significatifs en géométrie, cryptographie,... Un tel outil est égelement le complément indispensable des approches mixtes comme l'interfaçage avec des systèmes de calcul formel (Delahaye et Mayero 2005).

Cette bibliothèque de calcul va quant à elle nous servir de base pour programmer l'algorithme de CAD.

Chapitre 2

Vers la certification de l'algorithme des sous-résultants

Ce chapitre est consacré à la preuve formelle d'un algorithme de calcul de plus grand commun diviseur (pgcd) pour les polynômes à coefficients dans un anneau. Ceci constitue un problème classique du calcul formel, qui a donné lieu à une abondante littérature.

Le chapitre s'articule autour de trois points.

La première partie propose un *algorithme de division euclidienne* pour les polynômes en forme de Horner décrits dans le chapitre 1. L'originalité de cet algorithme est qu'il repose uniquement sur une induction structurelle grâce à la représentation de Horner.

Ensuite on généralise la notion de division euclidienne à celle de pseudo-division euclidienne, qui permet de définir des suites de (pseudo-)restes, comme celles de l'algorithme d'Euclide, même quand on divise des polynômes à coefficients dans un anneau.

Puis on introduit la notion de *déterminant polynomial*, qui permet de définir les polynômes sous-résultants. Pour formaliser ces déterminants polynomiaux, j'ai eu besoin de définir dans le système Coq des fonctions multilinéaires antisymétriques alternées qui prennent leurs arguments dans des anneaux.

Enfin, la dernière partie du chapitre est consacrée au théorème fondamental des sous-résultants. On montre grâce au *lemme de décalage* que la suite des polynômes sous-résultants, définis par déterminants polynomiaux forment en fait une suite de pseudo-restes et permet de calculer le pgcd de deux polynômes.

2.1 Calcul de pgcd polynomiaux en calcul formel

Le calcul efficace du plus grand commun diviseur de deux polynômes est un problème central du calcul formel. Dans un système de calcul formel, il est au coeur de nombreuses opérations courantes ayant trait aux simplifications d'expressions polynomiales ou de fractions rationnelles.

Les algorithme implémentés dans les systèmes de calcul formel se divisent principalement en quatre grandes familles.

Une première famille est constituée des algorithmes modulaires. Le fait de travailler sur des corps finis permet de limiter la croissance des coefficients que l'on manipule, ce qui est un des principaux enjeux de ce problème. Cette classe d'algorithmes repose sur des calculs de restes chinois, ou sur le lemme de Hensel (Geddes, Czapor, et Labahn 1992).

Les variantes du lemme de Hensel peuvent être vues comme un analogue de la méthode Newton pour les anneaux commutatifs complets et son basées sur des décompositions sur corps finis.

Il existe également des versions probabilistes très efficaces de ces algorithmes, comme dans les travaux de Zippel (Zippel 1993), qui proposent un algorithme d'interpolation pour le calcul du pgcd polynomial.

Une troisième famille repose sur des calculs de chaînes de (pseudo-)restes, dont la variante la plus efficace est celle des sous-résultants, qui fait l'objet de ce chapitre.

Enfin, les systèmes de calcul formel utilisent également des heuristiques qui permettent par exemple d'exploiter les factorisations triviales, ou bien les réductions à des problèmes de pgcd sur les entiers.

Savoir quel algorithme sera le plus efficace, même sur une instance particulière, est un problème difficile, et il n'existe pas de procédure de décision qui permette par exemple de lier une méthode de calcul à certain types de polynômes.

D'après l'étude comparative de Liao et Fateman (Liao et Fateman 1995), Maple applique des algorithmes modulaires en cas d'échec de son algorithme heuristique GCDHE, tandis que Macsyma et Mathematica utilisent une approche probabiliste à la Zippel, rattrapée en cas d'échec par un algorithme de sous-résultants.

En théorie, l'algorithme des sous-résultants n'est pas clairement meilleurs que ces autres approches et il manipule certainement des coefficients plus gros que dans les alternatives qui figurent à ce banc d'essai.

Néanmoins, il constitue une approche générale pour le problème, puisqu'il ne requiert sur l'ensemble des polynômes que l'existence du pgcd de deux éléments. Les algorithmes modulaires et d'interpolation sont destinés quant à eux à des cofficients à valeurs entières.

Enfin, l'approche des sous-résultants est une solution raisonnable puisque dans les tests d'implémentation de Liao et Fateman, cet algorithme arrive en première position dans presque la moitié des cas.

2.2 Division euclidienne, pseudo-division euclidienne

Dans tout ce chapitre, si D est un anneau commutatif intègre, on notera $deg(P)$ le degré d'un polynôme $P \in D[X]$. Le coefficient dominant d'un polynôme $P \in D[X]$ de degré d est celui porté par le monôme X^d. On le note $lcoef(P)$.

2.2.1 Fonctions partielles

On se propose de définir une opération de division euclidienne sur les polynômes en forme de Horner. Étant donnés deux polynômes P et Q dans cette représentation, on veut calculer, quand c'est possible, A et B deux nouveaux polynômes qui vérifient :

$$P = AQ + B \text{ avec } deg(B) < deg(Q)$$

En particulier, on ne calculera pas de tels A et B quand le diviseur Q est un polynôme nul. D'autre part, l'anneau commutatif C des coefficients est supposé intègre mais ne sera pas nécessairement un corps. Deux coefficients, même non nuls, ne sont donc pas nécessairement divisibles l'un par l'autre.

Par exemple si l'on considère des polynômes à coefficients entiers, on peut bien effectuer dans $\mathbb{Z}[X]$ la division de $X^2 + X + 1$ par $(X + 1)$:

$$X^2 + X + 1 = (X + 1)X + 1$$

Par contre, on ne peut diviser $X^2 + X + 1$ par $2X$ car le coefficient dominant 1 de $X^2 + X + 1$ n'est pas un multiple de 2, le coefficient dominant de $2X$.

En fait on programme la division euclidienne comme si on travaillait avec des coefficients dans un corps, mais si l'une des opérations de divisions sur les coefficients échoue, on fait échouer toute la division.

Il y a plusieurs façon d'envisager la programmation de cette fonction partielle. Dans ce cas, il m'a semblé que la solution la plus légère et la plus appropriée était de contourner le problème en faisant de cette division une fonction *totale*, et de repousser dans les spécifications l'expression du domaine de validité des propriétés de cette fonction. Un choix analogue avait été adopté et motivé dans la formalisation des nombres réels en HOL-Light (Harrison 1994).

Ainsi on va programmer une fonction `Pol_div_eucl` de type

`Pol_div_eucl : Pol C → Pol C → (Pol C) * (Pol C)`

dont la spécification est la suivante : pour tous polynômes P et Q, la fonction `Pol_div_eucl` calcule un couple de polynômes (A, B) tels que :

- soit $Q = 0$,
- soit $P = AQ + B$ avec $deg(B) < deg(Q)$
- soit $(A, B) = (0, 0)$.

On dira que P est un multiple de Q, de facteur A si :

- soit $P = 0$
- soit $A \neq 0$ et $B = 0$

La spécification de la division euclidienne, c'est à dire la preuve que l'identité

$$P = AQ + B \text{ avec } deg(B) < deg(Q)$$

est valide ne sera prouvée que sous les hypothèses que $(A, B) \neq (0, 0)$ ou bien $P = 0$.

Dans les autres cas, on sort du domaine de validité pour la fonction de division. La preuve de correction de l'algorithme de division assure seulement que les relations de divisibilités calculées sont correctes et non pas que *toutes* les relations de divisibilités existant dans l'anneau sont calculées.

2.2.2 Division euclidienne en représentation de Horner creuse

Il est aisé de programmer la division euclidienne des polynômes en forme de Horner (non creuse) de façon structurelle par rapport au dividende du fait que cette forme présente les coefficients de poids faible en tête et une structure simple. Mais cette représentation contient trop de zéros dans le cas des polynômes creux pour être raisonnable.

Le forme de Horner creuse est compacte : elle peut en fait être interprétée comme la liste ordonnée des coefficients non nuls, avec les coefficients de poids faible en tête.

De plus il est remarquable qu'elle conserve la possibilité de définir cette opération de division par récursion structurelle sur le dividende pour les polynômes en forme normale de Horner creuse, par le biais d'un point fixe local.

De fait, la forme de Horner creuse ressemble beaucoup à une représentation de nombres dans une "base formelle" X. Par "base formelle" on entend ici une base qui serait infinie au sens ou on ne propage pas de retenue. L'algorithme de division que je propose ci-dessous est proche de celui qui permet de diviser des nombres entiers, par exemple en représentation binaire comme dans la bibliothèque standard de Coq.

Dans tout ce qui suit, on utilise pour les polynômes en forme de Horner creuse les définitions et notations introduite au chapitre 1.

Division d'un polynôme par une constante

Ici on veut diviser un polynôme arbitraire `P` par un polynôme constant `(Pc c)`. En fait on va seulement propager l'opération de division par `c` sur tous les coefficients, mais en retournant le polynôme nul si l'une de ces divisions échoue.

La division sur les coefficients est notée `//` et le test à zéro sur les coefficients est noté `czero_test`.

On fait une récurrence sur le polynôme `A` et on divise par la constante `q`.

On rappelle que le polynôme nul est note `P0`.

```
Fixpoint Pol1_div_cst(A:Pol)(q:Coef){struct A}: Pol:=
     if (Pol_zero_test A) then P0
     else
     match A with
       |Pc a ⇒ Pc (a // q)
       |PX P i p ⇒
         let P':= (Pol1_div_cst P q) in
         if (Pol_zero_test P') then P0 else
           let p':= p // q in
           if czero_test p' then P0 else
             PX P' i p
     end.
```

Division euclidienne par un polynôme non constant

On programme maintenant la division d'un polynôme quelconque A par un polynôme $B = B_1X^j + b$ dont on sait qu'il est de degré non nul. On rend deux polynômes Q et R, contenant le quotient et le reste de la division. On procède par induction sur la structure du dividende A.

- Soit A est constant.

  On rend $Q = 0, R = A$.

- Soit A est de la forme $A_1X^i + a$. On cherche à calculer Q et R tels que :

  $$A_1X^i + a = (B_1X^j + b)Q + R \text{ avec } deg(R) < deg(B_1) + j$$

  On calcule récursivement Q_1 et R_1 tels que :

  $$A_1 = (B_1X^j + b)Q_1 + R_1 \text{ avec } deg(R_1) < deg(B_1) + j$$

  D'où :

  $$A_1X^i + a = (B_1X^j + b)Q_1X^i + R_1X^i + a$$

- **1er cas :** $deg(R_1) + i < deg(B_1) + j$ ou bien $R_1 = 0$.
  On rend $Q = Q_1 X^i$ et $R = R_1 X^i + a$.
- **2ème cas :** $deg(R_1) + i \geq deg(B_1) + j$ et $R_1 \neq 0$
  On suppose qu'on dispose d'une fonction auxiliaire div_aux qui prend en argument un polynôme de la forme SX^i et un polynôme non constant $CX^k + c$ tels que :
  - $deg(S) < deg(C) + k$
  - $S \neq 0$

  et calcule le quotient et le reste de la division euclidienne de SX^i par $CX^k + c$.
  La fonction div_aux permet de calculer Q_2 et R_2 tels que :

$$R_1 X^i = (B_1 X^j + b)Q_2 + R_2 \text{ avec } deg(R_2) < deg(B_1) + j$$

  On rend $Q = Q_1 X^i + Q_2$ et $R = R_2 + a$

Reste à expliciter la fonction auxiliaire div_aux. Étant donnés deux polynômes S et C, un coefficient c et deux entiers i et k, tels que :
- (1) $deg(S) < deg(C) + k$
- (2) $S \neq 0$

on veut calculer deux polynômes Q et R tels que :

$$SX^i = (CX^k + c)Q + R \text{ avec } R < deg(C) + k$$

Le calcul se fait par récursion structurelle sur l'exposant i qui est un entier en base 2 représenté en Coq par le type de de données du code 1.3. Cette récursion repose sur la distinction de trois cas pour un entier i en base 2 : soit $i = 1$, soit i est de la forme $i = 2p$ pour un entier p en base 2 , soit i est de la forme $i = 2p + 1$ pour un entier p en base 2.
- **Cas 1 :** $deg(S) + i < deg(C) + k$
  On rend $Q = 0$ et $R = SX^i$
- **Cas 2 :** $deg(S) + i \geq deg(C) + k$ et $i = 1$.
  Alors $deg(S) + 1 = deg(C) + k$ vu (1). Il existe un coefficient $\lambda \in C$ et un polynôme $R \in C[X]$ tel que :

$$SX = \lambda(CX^k + c) + R \text{ avec } deg(R) < deg(C) + k$$

  On rend $Q = \dfrac{coefdom(C)}{coefdom(S)}$ et $R = S - (CX^k + c)Q$
- **Cas 3 :** $deg(S) + i \geq deg(C) + k$ et $i = 2p$
  On calcule récursivement Q_1 et R_1 tels que :

$$SX^p = (CX^k + c)Q_1 + R_1 \text{ avec } deg(R_1) < deg(C) + k$$

  - **Cas 3.1 :** $deg(R_1) + p < deg(C) + k$ ou bien $R_1 = 0$
    On rend $Q = Q_1 X^p$ et $R = R_1 X^p$
  - **Cas 3.2 :** $deg(R_1) + p \geq deg(C) + k$ et $R_1 \neq 0$
    On calcule récursivement Q_2 et R_2 tels que :

$$R_1 X^p = (CX^k + c)Q_2 + R_2 \text{ avec } deg(R_2) < deg(C) + k$$

    On rend $Q = Q_1 X^p + Q_2$ et $R = R_2$

– **Cas 4 :** $deg(S) + i \geq deg(C) + k$ et $i = 2p + 1$
On calcule récursivement Q_1 et R_1 tels que :

$$SX^p = (CX^k + c)Q_1 + R_1 \text{ avec } deg(R_1) < deg(C) + k$$

– **Cas 4.1 :** $degR_1 + p + 1 < deg(C) + k$
On rend $Q = Q_1X^{p+1}$ et $R = R_1X^{p+1}$

– **Cas 4.2 :** $degR_1 + p + 1 \geq deg(C) + k$
On calcule récursivement Q_2 et R_2 tels que :

$$R_1X^p = (CX^k + c)Q_2 + R_2 \text{ avec } deg(R_2) < deg(C) + k$$

– **Cas 4.3 :** $deg(R_2) + 1 < deg(C) + k$
On rend $Q = Q_1X^{p+1} + Q_2$ et $R = R_2$

– **Cas 4.4 :** $deg(R_2) + 1 \geq deg(C) + k$
Alors $deg(R_2) + 1 = deg(C) + k$ il existe $\lambda \in C$ et un polynôme R_3 tels que :

$$R_2X = \lambda(CX^k + c) + R \text{ avec } deg(R_3) < deg(C) + k$$

En fait $\lambda = \dfrac{coefdom(C)}{coefdom(R_2)}$ et
On rend $Q = Q_1X^{p+1} + Q_2 + \lambda$ et $R = R_3 = R_2 - \lambda(CX^k + c)$

2.2.3 Preuve formelle des spécifications des opérations de division

Les algorithmes détaillés dans la section précédente présentent la particularité d'utiliser intensivement les opérations de test à zéro, aussi bien dans la structure de coefficients que dans celle des polynômes.

Dans un souci d'efficacité, suivant la discipline commune à tout le développement selon laquelle les polynômes arguments et résultats de toutes les opérations sont en forme normale, le test à zéro sur les polynômes se programme ainsi :

```
Definition Pol_zero_test(P:Pol):bool:=
  match P with
    |Pc c ⇒ (czero_test c) (* test sur la structure de coefs *)
    |PX _ _ _⇒ false (* non constant ⇒ non nul *)
  end.
```

Code 2.1: *Test à zéro des polynômes*

Néanmoins, pour spécifier ce test, on aura besoin de spécifier ses conditions de validité, et en particulier, sous cette forme, le test à zéro n'est pas un morphisme pour l'égalité de setoïde avec laquelle nous travaillons (voir section 1.1.4).

C'est pourquoi il est nécessaire de définir formellement les conditions de forme normale qui détermineront les conditions de validité de tous les énoncés de correction.

```
Inductive Pol_nf : Pol → Prop:=
(* Un polynome constant est toujours en forme normale *)
  |Pc_nf : ∀ c, Pol_nf (Pc c)
(* cX^i + p est en forme normale si c ≠ 0 *)
  |PX_Pc_nf : ∀ c i p,
    czero_test c = false → Pol_nf (PX (Pc c) i p)
(* (PX^i + p)X^j + c est en forme normale si
    - PX^i + p est en forme normale
    - p ≠ 0 *)
  |PX_PX_nf : ∀ P i p j c,
    Pol_nf (PX P i p) →
    czero_test p = false
    → Pol_nf (PX (PX P i p) j c).
```

Code 2.2: *La propriété de forme normale*

Il sera pratique pour pouvoir réécrire commodément les arguments du test à zéro, de définir une deuxième version `Pol_zero_test2` de cette fonction de test booléen `Pol_zero_test` , qui est elle-même un morphisme. Cette nouvelle fonction est moins efficace que la première car elle doit effectuer plus de tests du fait qu'elle ne suppose pas que son argument est en forme normale. Par contre c'est un morphisme, et elle coïncide avec la version destinée aux calculs sur l'ensemble des polynômes en forme normale.

```
Fixpoint Pol_zero_test2(P:Pol):bool :=
  match P with
    |Pc c ⇒ czero_test c
    |PX P i p ⇒ andb (Pol_zero_test2 P) (czero_test p)
  end.
```

Code 2.3: *Test à zéro des polynômes version preuves*

L'implémentation des opérateurs de division repose sur l'opération de division `cdiv` sur l'ensemble des coefficients.

On travaille donc en supposant que cet opérateur satisfait les spécifications suivantes :

```
(* Si le diviseur d et le dividende x/d sont non nuls, alors x =
  (x/d)*d *)
Hypothesis cdiv_spec1 : ∀ d,
  czero_test d = false →
  ∀ x,
  czero_test (x // d) = false →
  x == (x//d) ** d.

(* On a toujours 0 = (0/x) * x.*)
(*  Ce cas englobe le cas degenere du diviseur nul *)
Hypothesis cdiv_spec2 : ∀ d x,
  czero_test x = true → x  == (x//d) ** d.
```

Les théorèmes de correction des trois opérateurs de division utilisés énoncent que pour des polynômes en forme normale, les propriétés énoncées à la fin de la section 2.2.1 sont

valides. Leur preuve formelle est en cours de développement, en collaboration avec L. Pottier.

2.2.4 Pseudo-divisions, Chaînes de pseudo-restes

Dans le cas où l'ensemble des coefficients d'un anneau de polynômes est un anneau commutatif intègre (et non nécessairement un corps), on peut tout de même généraliser la division euclidienne ci-dessus. En fait si D est un anneau commutatif intègre, et si $P, Q \in D[X]$, dans le déroulement de la division euclidienne de P par Q, les seuls dénominateurs qui peuvent être introduits dans les opérations sur les coefficients sont des puissances du coefficient dominant de Q.

De fait il suffit de multiplier P par $lcoef(Q)^{deg(P)-deg(Q)+1}$ pour être sûr que la division euclidienne des polynômes sera exacte, c'est à dire qu'on calcule bien un quotient et un reste qui satisfont la propriété :

$$P = AQ + B \text{ avec } deg(B) < deg(Q)$$

Cette propriété avait déjà été observée par Jacobi (Jacobi 1836), c'est pourquoi on donne son nom à ce facteur.

Dans toutes les définitions qui suivent, D est un anneau factoriel, ce qui permet en particulier d'assurer l'existence de plus grands communs diviseurs.

Premières définitions

Définition 2.2.1 (Facteur de Jacobi) *Soit $P, Q \in D[X]$ deux polynômes. On appelle* facteur de Jacobi de la division de P par Q *la quantité :*

$$lcoef(Q)^{deg(P)-deg(Q)+1}$$

On a donc un moyen de définir une opération de division exacte pour tout couple de polynômes.

Définition 2.2.2 (Pseudo-division) *Soit $P = p_0 + \ldots p_n X^n$ et $Q = q_0 + \ldots q_m X^m$ deux polynômes éléments de $D[X]$, avec $p_n, p_m \neq 0$ et $n \geq m$.*

Le reste (resp. quotient) de la division $q_m^{n-m+1} P$ par Q est appelé le pseudo-reste *(resp.* pseudo-quotient*) de P par Q et est noté* preste *(resp.* pquo*). Cette opération est appelé* pseudo-division *de P par Q. Ces deux polynômes sont uniques et sont des éléments de $D[X]$.*

Exemple 2.2.4.1 *On considère des polynômes dans $\mathbb{Z}[X]$.*
- *$P = X^2$ et $Q = 2X + 1$: $pquo(P, Q) = 2X - 1$ et $preste(P, Q) = 1$*
- *$P = 2X^2 + 2X$ et $Q = 2X + 2$: $pquo(P, Q) = 4X$ et $preste(P, Q) = 0$*

Définition 2.2.3 (Similarité) *Soit $P, Q \in D[X]$, P et Q sont* similaires *($P \sim Q$) s'il existe $a, b \in D$ tels que $aP = bQ$.*

Exemple 2.2.4.2 *Toujours dans le cas de $\mathbb{Z}[X]$, les polynômes $2X + 2$ et $3X + 3$ sont similaires. Les polynômes $X + 1$ et $X + 2$ ne sont pas similaires.*

L'algorithme d'Euclide calcule le plus grand commun diviseur (noté pgcd) de deux polynômes à coefficients dans un corps en effectuant une succession de divisions euclidiennes. On peut généraliser cet algorithme au cas où les coefficients sont dans une structure d'anneau. Il suffit de remplacer chaque étape de division euclidienne par une étape de *pseudo*-division euclidienne. Cependant, comme le facteur de Jacobi que l'on introduit pour rendre chaque division exacte n'est pas nécessairement le facteur correctif le plus petit possible, on autorise des factorisations scalaires à chaque étape de l'algorithme et on ne demande dans la chaîne des polynômes calculés que des relations de similarité aux pseudo-restes.

Définition 2.2.4 (Chaîne de pseudo-reste) *Soit $F_1, F_2 \in D[X]$, avec $deg(F_1) > deg(F_2)$. Soit $F_1 \dots F_k \in D[X]$ une suite de polynômes non nuls tels que :*

$$F_i \sim preste(F_{i-2}, F_{i-1}) \text{ pour } i = 3 \dots k \quad et \quad preste(F_{k-1}, F_k) = 0$$

Cette suite est appelée chaîne de pseudo-restes.
De la définition ci-dessus se déduit immédiatement :

$$\forall i = 3 \dots k, \quad \exists \alpha_i, \beta_i \in D \text{ and } \exists Q_i \sim pquo(F_{i-2}, F_{i-1}) \text{ tel que}$$
$$\beta_i F_i = \alpha_i F_{i-2} - Q_i F_{i-1} \quad deg(F_i) < deg(F_{i-1})$$

Le sens informel de ces coefficients de similarité α et β est le suivant : α assure que la division euclidienne sera possible en restant dans $D[X]$, et β est un facteur scalaire que l'on peut éliminer du reste.

Définition 2.2.5 (Contenu, partie primitive) *Soit $P \in D[X]$ le* contenu *de P, noté $cont(P)$ est un plus grand commun diviseur de ses coefficients. Il est unique à multiplication près par un inversible de D.*

Si les coefficients de P sont premiers entre eux, alors P est dit primitif. *Si ce n'est pas le cas, sa* partie primitive, *notée $pp(P)$ est définie par $P = cont(P)pp(P)$.*

Le pgcd de deux éléments de $D[X]$ est le produit de leurs contenus par le pgcd de leurs parties primitives. De plus, si deux polynômes F_1 et F_2 de $D[X]$ sont primitifs, alors leur pgcd est aussi primitif. Ainsi, avec les notations de la définition 2.2.4, pour tous polynômes F_1 et F_2, $gcd(F_1, F_2) = pp(F_k)$.

Contrôler la croissance des coefficients

Dans toute la suite, on considère $F_1 = p_0 + \dots p_n X^n$ et $F_2 = q_0 + \dots q_m X^m$, deux polynômes dans $D[X]$ et $(F_i)_{i=1\dots k}$ une chaîne de pseudo-restes telle que pour $i = 3 \dots k$:

$$\beta_i F_i = \alpha_i F_{i-2} - Q_i F_{i-1} \quad deg(F_i) < deg(F_{i-1}) \quad (1)$$

On note n_i le degré de F_i et c_i le coefficient dominant de F_i pour $i = 3 \dots k$. On appelle (1) une *relation pseudo-euclidienne.*

Calculer le pgcd de deux polynômes P et Q, c'est calculer le dernier polynômes d'une chaîne de pseudo-reste s issue de P et Q. On veut pouvoir effectuer ce calcul le plus

efficacement possible. Le choix des facteurs de similarité α_i et β_i permet de jouer sur la taille des coefficients des polynômes dans la chaîne.

Une façon naïve de calculer une chaîne de pseudo-restes est de poser $F_i = preste(F_{i-2}, F_{i-1})$. Cette chaîne de pseudo-restes est appelée *chaîne euclidienne de pseudo-restes*, d'après la terminologie de G.E. Collins (Collins 1967). Malheureusement, cette définition peut conduire à une explosion de la taille des coefficients dans les polynômes de chaîne.

En fait il existe même une borne inférieure polynomiale, donnée par C. K. Yap (Yap 2000) et D.E. Knuth décrit ce phénomène de la manière suivante (Knuth 1998) : "Ainsi la borne supérieure [...] serait approximativement de $N^{0,5^{2,414^n}}$ et les expérimentations montrent que cet algorithme simple a en fait exactement ce comportement : le nombre de chiffres dans les coefficients croît exponentiellement à chaque étape !". Néanmoins la remarque de J. von zur Gathen et T. Lücking (von zur Gathen et Lücking 2003) permet d'espérer de meilleurs résultats dans le cas qui nous occupe : "Dans une unique division, dont les arguments sont choisis aléatoirement, on ne peut espérer beaucoup mieux que la pseudo-division de Jacobi pour maintenir un reste à coefficients entiers. Mais les résultats de l'algorithme d'Euclide sont tellement dépendants qu'il est toujours possible d'extraire des facteurs importants."

La stratégie opposée consiste à choisir $F_i = pp(preste(F_{i-2}, F_{i-1}))$, et ce choix minimise la croissance de la taille des coefficients. Cette chaîne de pseudo-restes est appelée *chaîne primitive de pseudo-restes*, toujours en suivant la terminologie de Collins. Mais l'inconvénient majeur de cette stratégie est qu'elle suppose des calculs récursifs de pgcd à chaque étape du calcul de la chaîne, ce qui est général bien trop coûteux. L'un des cas où ce choix est très mauvais est par exemple le cas où on travaille avec des polynômes à plusieurs variables, vus comme éléments de $D[X_1] \dots [X_k]$.

La solution que nous proposons de formaliser est appelée *chaîne des sous-résultants*. Elle constitue un compromis entre les deux solutions précédentes : à chaque étape, on *prédit* à la fois un facteur α, suffisant pour permettre la division euclidienne, et un coefficient β de taille significative qu'on va pouvoir factoriser dans le reste. De plus le calcul de ces deux facteurs se fait avec une complexité satisfaisante et permet de contrôler la croissance des coefficients.

Ci-après on reproduit des exemples construits par J. von zur Gathen et T. Lücking (von zur Gathen et Lücking 2003) pour comparer les trois définitions de chaînes de pseudo-restes que nous avons citées. Dans ces exemples, les α_i seront toujours les facteurs de Jacobi de la pseudo division, et les β_i sont les facteurs qui peuvent être extraits des pseudo-restes selon l'algorithme utilisé.

i	α_i	β_i	F_i
1			$9X^6 - 27X^4 - 27X^3 + 72X^2 + 18X - 45$
2			$3X^4 - 4X^2 - 9X + 21$
3	$3^3 = 27$	1	$-297x^2 - 729X + 1620$
4	-26198073	1	$3245333040X - 4899708873$
5	10532186540515641600	1	-1659945865306233453993

Chaîne euclidienne : pas de factorisation ($\beta_i = 1$), une croissance exponentielle des coefficients

i	α_i	β_i	F_i
1			$9X^6 - 27X^4 - 27X^3 + 72X^2 + 18X - 45$
2			$3X^4 - 4X^2 - 9X + 21$
3	$3^3 = 27$	3	$-11X^2 - 27X + 60$
4	-1331	9	$18320X - 27659$
5	335622400	1959126851	-1

Chaîne primitive : factorisations optimales, mais calculs récursifs coûteux

i	α_i	β_i	F_i
1			$9X - 27X^4 - 27X^3 + 72X^2 + 18X - 45$
2			$3X - 4X^2 - 9X + 21$
3	27	3	$297X^2 + 729X - 1620$
4	26198073	-243	$13355280X - 20163411$
5	178363503878400	2910897	9657273681

Chaîne des sous-résultants : le compromis. On supprime des facteurs significatifs, mais moins importants que dans le cas primitif. Par contre calculer ces facteurs β_i est beaucoup moins coûteux.

2.3 Déterminants polynomiaux

Avant de définir précisément les polynômes dits sous-résultants qui composent la chaîne du même nom, on introduit une définition fondamentale pour la suite du développement. Dans tout ce qui suit, D est un anneau commutatif intègre et on note $\mathcal{F}_n \subset D[X]$ le module libre de type fini des polynômes de degré strictement plus petit que n, qui est engendré par la base monomiale :

$$\mathcal{B} = X^{n-1} \dots X, 1$$

2.3.1 Définition

Proposition 2.3.1 (Déterminant polynomial) *Soit n et m deux entiers tels que $0 < m \leq n$. Il existe une unique application multilinéaire alternée $pdet_{n,m} : (\mathcal{F}_n)^m \to \mathcal{F}_{n-m+1}$ qui satisfait pour tous $n > i_1 > \cdots > i_{m-1} > i$:*

$$\begin{cases} pdet_{n,m}(X^{i_1}, \dots, X^{i_{m-1}}, X^i) = X^i & \text{si pour tout } j < m, \quad i_j = n - j \quad (i) \\ pdet_{n,m}(X^{i_1}, \dots, X^{i_{m-1}}, X^i) = 0 & \text{sinon} \qquad\qquad\qquad\qquad\quad (ii) \end{cases}$$

Cette application est appelée déterminant polynomial *d'une famille de m polynômes de $\mathcal{F}_n$.*

Cette définition s'illustre en termes de déterminants de matrices dont les coefficients sont ceux des polynômes en arguments de l'application multilinéaire.

Soit $\mathcal{P} = \{P_1, \ldots, P_m\} \subset \mathcal{F}_n$ une famille de m polynômes, où

$$P_i = p_0^i + \cdots + p_{n-1}^i X^{n-1}$$

On note $\mathbf{M}(\mathcal{P})$ la matrice (éventuellement non carrée) de la famille $\mathcal{P}$ dans la base $\mathcal{B}$.

$$\mathbf{M}(\mathcal{P}) = \begin{bmatrix} p_{n-1}^1 & \cdots & p_{n-1}^m \\ \vdots & & \vdots \\ p_0^1 & \cdots & p_0^m \end{bmatrix}$$

Preuve L'unicité de l'application *pdet* s'obtient par antisymétrie et multilinéarité après avoir décomposé chacun des arguments sur la base $\mathcal{B}$.

On note $\mathbf{M}_{n,m}(\mathcal{P})$ la matrice carrée dont les $m-1$ premières lignes sont celles de $\mathbf{M}(\mathcal{P})$ et la dernière est formée des polynômes $P_1, \ldots, P_m$.

$$\mathbf{M}_{n,m}(\mathcal{P}) = \begin{bmatrix} p_{n-1}^1 & \cdots & p_{n-1}^m \\ \vdots & & \vdots \\ p_{n-m+1}^1 & \cdots & p_{n-m+1}^m \\ P_1 & \ldots & P_m \end{bmatrix}$$

On définit $pdet_{n,m}$ comme étant $det(\mathbf{M}_{n,m}(\mathcal{P}))$, où det est le déterminant usuel des matrices carrées, dont les coefficients sont ici des polynômes.

Dans cette définition, la première condition (i) correspond au déterminant d'une matrice triangulaire inférieure :

$$pdet_{n,m}(X^{n-1}, \ldots, X^{n-m+1}, X^i) = det \begin{bmatrix} 1 & & & 0 \\ 0 & & 0 & \\ \vdots & \ddots & & \vdots \\ 0 & & 1 & 0 \\ X^{n-1} & \ldots & X^{n-m+1} & X^i \end{bmatrix}$$

La deuxième condition (ii) correspond au déterminant d'une matrice triangulaire inférieure qui a au moins un zéro sur sa diagonale. □

2.3.2 Formalisation des déterminants

La contribution Algebra au système Coq (Pottier 1999) a servi de base à J. Stein pour son développement d'une bibliothèque d'algèbre linéaire LinAlg (Stein 2003). Cette bibliothèque contient déjà suffisamment de matériau pour pouvoir être à son tour augmentée et servir de base à une bibliothèque d'algèbre multilinéaire.

Ce n'est pas la solution que j'ai adoptée, pour plusieurs raisons. La première est que la théorie mathématique qui sous-tend une définition générale des déterminants polynomiaux est celle des modules libres de type fini, alors que LinAlg concerne les structures d'espaces vectoriels. D'autre part, lorsque j'ai envisagé ce travail, la bibliothèque LinAlg était encore en cours de développement et donc pas stabilisée. Enfin, dans le cas qui nous occupe, la théorie générale de la multilinéarité et des modules de type fini n'est pas un objectif et représente un travail conséquent, beaucoup plus général que l'utilisation que nous faisons ici de *pdet*.

Je propose donc de définir une version calculatoire de déterminants, de la manière suivante.

Pré-requis :

- Un ensemble de coefficients scalaire `C` muni d'une structure d'anneau commutatif intègre
- Un ensemble de vecteurs `V` muni d'une structure d'anneau commutatif intègre.
- La dimension `dim` de `V` sur `C`.
- Un opération de multiplication par un coefficient scalaire `scal : C → V → V`.
- Une fonction de projection sur une coordonnée `phi : nat → nat → V → V`.

L'interprétation de ces paramètres est la suivante :

- `V` est un module libre de type fini sur l'anneau `C`.
- Le type de `phi` devrait plutôt être `nat → nat → V → C`, mais on suppose ici qu'on injecte `C` dans `V` pour les besoins du déterminant polynômial. Un développement analogue *mutatis mutandis* serait possible pour obtenir un déterminant à valeurs dans `C` en choisissant `phi : nat → nat → V → C`.
- En pratique, la valeur de phi sera une coordonnée pondérée par la polarité de la ligne par rapport à laquelle on développe le déterminant : la première ligne est de polarité 1, la deuxième de polarité -1... (voir section 3.2.3)
- Si $(e_1, \ldots, e_d)$ est une base de `V`, la fonction `phi k i v` calcule la i-ème coordonnée du vecteur `v` projeté sur $(e_1, \ldots, e_k)$. Elle rend zéro si `i > d`.

Définition :

On définit le déterminant d'une liste de vecteurs par son développement par rapport à une ligne. Soit `l` une liste de vecteurs de longueur `n`, on suppose que `n` est plus petit que `d` la dimension de `V` sur `C`. On calcule le déterminant de la famille des projetés des vecteurs de `l` sur $(e_1, \ldots, e_n)$.

Ce calcul se fait en deux temps, par récursion sur la dimension `n` du déterminant (qui est la taille de la liste).

- Par convention, le déterminant d'une liste vide vaut 1.
- Sinon, si la longueur de la liste est `n+1`, on calcule le déterminant en le développant par exemple par rapport à la dernière ligne. C'est possible si d'une part on sait calculer les déterminants d'ordre `n`, pour calculer les cofacteurs qui apparaissent dans le développement, et d'autre part si on sait extraire des vecteurs la coordonnée

qui se trouve sur cette avant-dernière ligne, grâce à la fonction `phi`

Cette récursion est réalisée par la fonction `det_aux`, décrite dans le code 2.4

```
Fixpoint det_aux (n: nat) (l: list V) {struct n}: V :=
 match n with
   O ⇒ 1
 | S n1 ⇒ rec_det (phi dim n) (det_aux n1) l nil
 end.
```

Code 2.4: *Mise en place de la récursion sur la dimension du déterminant*

Il faut maintenant programmer le calcul d'un déterminant de dimension `n+1` lorsqu'on dispose d'une fonction `f` qui extrait la coordonnée de chaque vecteur (colonne) qui correspond à la ligne par rapport à laquelle on développe et d'une fonction `rec` qui calcule les déterminants d'ordre `n`. Dans `l2`, on place successivement les vecteurs dont on déjà extrait la coordonnée pertinente et calculé son cofacteur, `l1` contient les vecteurs qui restent à traiter.

La fonction `rec_det` du code 2.5 calcule cette somme alternée par récursion sur la structure de la liste `l1`.

```
Fixpoint rec_det
      (f: V → C) (rec: list V → V) (l1 l2: list V){struct l1}: V :=
  match l1 with
  | nil ⇒  P0
  | a:: l3 ⇒
      f a * rec (app l2 l3) - rec_det f  rec  l3 (app l2 (a::nil))
  end.
```

Code 2.5: *Calcul de la somme alternée*

Finalement on définit le calcul du déterminant d'une famille de vecteurs comme dans le code 2.6.

```
Definition det l :=  det_aux (length l) l.
```

Code 2.6: *Définition du déterminant*

Dans ce qui suit, on note `==` l'égalité sur `V`, et `!*` est une notation infixe pour le produit scalaire `scal`. Les opérations d'anneau sont notées par les symboles usuels. L'opérateur de concaténation des listes est noté `app`.

En travaillant directement sur les définitions des codes 2.4, 2.5 et 2.6, on montre les propriétés attendues pour le déterminant :

– Linéarité par rapport au premier élément de la liste :

```
Theorem det_lin : ∀ a b c d l,
 det ((add (a !* b) (c !* d)) :: l)
                  ==
 a !* det (b :: l) + c !* det (d :: l).
```

– Antisymétrie :

```
Theorem det_antisym : ∀ a b l1 l2 l3,
det (app l1 (app  (a :: l2)  (b :: l3)))
                  ==
- det (app l1 (app   (b :: l2)  (a :: l3))).
```

– Alternance :

```
Theorem det_alt: ∀ a l1 l2 l3,
   det (app l1 (app  (a :: l2)  (a :: l3))) == 0.
```

Nous aurons également besoin pour la suite du développement de la propriété suivante sur le développement des matrices semi-diagonales illustrée sur la figure 2.1

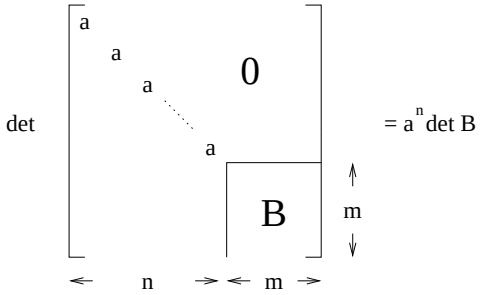


FIG. 2.1: *Déterminant des matrices semi-diagonales*

Cette propriété est plus délicate à obtenir que les précédentes car elle mêle dans la même expression des déterminants de différentes dimensions. Lors de cette preuve dans le système Coq, la manipulation de notre formalisation des déterminants a également mis à jour des choix inadaptés à nos besoins dans la récente implémentation de la réécriture sétoïde (Sacerdoti Coen 2006). Ces difficultés ont rendu cette dernière preuve formelle extrêmement fastidieuse.

2.3.3 Un premier exemple : déterminants de matrices à coefficients entiers

Dans cette section on met en oeuvre la formalisation présentée dans la section précédente afin de calculer les déterminants à coefficients entiers.

Les prérequis sont instanciés de la façon suivante :

– L'anneau des coefficients est l'anneau des entiers $\mathbb{Z}$.
– On se donne une dimension d pour les matrices dont on veut calculer les déterminants.
– L'ensemble des vecteurs est représenté par les listes d'entiers de longueur d. Il est important de rappeler qu'on ne demande pas ici de preuve de la longueur des listes. Les calculs ne seront corrects que si toutes les listes données en arguments du déterminant sont de longueur d.
– La multiplication scalaire d'un entier z par une liste l est la propagation sur tous les éléments de la liste du produit par z.
– L'implémentation de la fonction coordonnée détermine la ligne du déterminant par rapport à laquelle on développe.

Par exemple, le développement d'un déterminant de taille `d` par rapport à sa dernière ligne sera réalisée par la fonction `phi` décrite par le code 2.7. L'appel `nth k l a` calcule le k-ième élément de la liste l, numérotés de 0 à $|l|-1$, où $|l|$ est la longueur de la liste. Cet appel renvoie par défaut la valeur a quand $k \geq |l|-1$.

```
Definition phi d n l := (-1)^(n+1) * nth (n - 1)  l 0.
```

Code 2.7: *Développement par rapport à la dernière ligne*

Le facteur `(-1)^(n1)+` reflète la polarité de la dernière ligne de la matrice, qui est une matrice carrée de taille n. Elle donne le signe dont est affecté le premier élément de la dernière ligne de la matrice lorsqu'on développe par rapport à cette dernière.

Si maintenant on veut calculer le déterminant d'une matrice en utilisant un développement par rapport à la première ligne de la matrice, on pourra utiliser l'implémentation de `phi` décrite par le code 2.8.

```
Definition phi d n l := nth (d - n)  l 0.
```

Code 2.8: *Développement par rapport à la première ligne*

Cette fois la polarité de la première ligne étant toujours positive, on ne rajoute pas de facteur de signe dans phi.

2.3.4 Spécialisation de la formalisation pour les déterminants polynomiaux

La motivation de la formalisation précédente est de pouvoir raisonner en Coq sur les déterminants polynomiaux. On rappelle les notations de la preuve de la propriété 2.3.1, selon lesquelles étant donnée une famille $\mathcal{P} = \{P_1, \ldots P_m\} \subset \mathcal{F}_n[X]$, le déterminant polynomial est le déterminant de la matrice suivante dans laquelle on identifie les coefficients à des polynômes constants.

$$\mathbf{M}_{n,m}(\mathcal{P}) = \begin{bmatrix} p^1_{n-1} & \cdots & p^m_{n-1} \\ \vdots & & \vdots \\ p^1_{n-m+1} & \cdots & p^m_{n-m+1} \\ P_1 & \ldots & P_m \end{bmatrix}$$

Développer le déterminant de cette matrice par rapport à son avant-dernière ligne permet une définition récursive du déterminant $pdet_{n,\square}$ en fonction de $pdet_{n-1,\square}$.

On utilise la formalisation de la section 2.3.2 pour définir *pdet* en Coq. Voici comment sont instanciés les pré-requis :

- Le type des coefficients est `Coef`, celui des coefficients des polynômes.
- Le type des vecteurs est `Pol`, celui des polynômes.
- La dimension de l'espace que l'on notera dorénavant `deg` est le degré maximal des polynômes considérés.
- Revenons à la définition de `det_aux` figurant sur le code 2.4. Pour une matrice de taille `n+1`, c'est la fonction `phi deg (n+1) : Pol → Pol` qui définit la ligne

par rapport à laquelle le déterminant est développé. Dans notre cas, si la famille $\mathcal{P}$ contient n polynômes dont `P`, l'application `phi deg n P` doit calculer la coordonnée de l'avant dernière ligne de la matrice $\mathbf{M}_{n,m}(\mathcal{P})$ dans la colonne qui correspond au polynôme `P`. De plus il faut pondérer cette coordonnée par un signe, qui est la polarité du coefficient dans la matrice.

Plus précisément, voici l'implémentation en Coq de la fonction `phi` qui permet de définir les déterminants polynomiaux. La notation `P0` désigne le polynôme constant nul, et la fonction `(get_coef k Q)` calcule la coordonnée du polynôme `Q` sur le monôme de degré `k`.

```
Definition phi (deg n:nat) :=
(* 1er cas : les colonnes sont forcement liees *)
if n ≥ deg +2 then (fun P: Pol ⇒ P0)
(* 2eme cas : n ≤ deg + 1 *)
else
   match n with
  (* n'est jamais appele : *)
    | O ⇒ (fun P: Pol ⇒ P0)
  (* une matrice de taille 1 ne contient qu'un polynome *)
    | 1 ⇒ (fun P: Pol ⇒ P)
  (* le cas general : le coefficient deg − (n −2) *)
    | _ ⇒
        if (is_even n) then
          (fun P : Pol ⇒ Pc (get_coef (deg - (n-2)) P))
        else
          (fun P : Pol ⇒ Pc (- get_coef (deg - (n-2)) P))
  end.
```

Code 2.9: *L'implémentation de phi pour les déterminants polynomiaux*

Cette formalisation des déterminants polynomiaux nous donne tout le matériel nécessaire pour pouvoir définir en Coq les *polynômes sous-résultants*. Les propriétés de ces polynômes sont dérivées à partir des propriétés de ces déterminants particuliers.

2.4 Polynômes sous-résultants

2.4.1 Définition

Dans tout ce qui suit, on suppose que D est un anneau commutatif intègre.

Définition 2.4.1 (Polynômes sous-résultants) *Soit $P, Q \in D[X]$ deux polynômes tels que $deg(P) = n$ et $deg(Q) = m$, avec $n > m$. Alors pour $i = 0 \ldots n$, on définit le i-ème polynôme sous-résultant par :*

- $S_n(P, Q) = P$
- $S_i(P, Q) = 0$ *pour* $m < i < n$
- $S_i(P, Q) = S_i(P, Q) = pdet_{n+m-i,n+m-2i}(X^{m-i-1}P, \ldots, XP, P, Q, XQ, \ldots, X^{n-i-1}Q)$ *sinon*

Notons $P = p_0 + \cdots + p_n X^n$ et $Q = q_0 + \cdots + q_m X^m$. Si l'on revient à la définition par déterminants de matrices de *pdet*, on observe que lorsque la matrice $\mathbf{M}_i$, carrée d'ordre $n + m - 2i$, est bien définie :

$$S_i(P,Q) = det\ \mathbf{M}_i \text{ avec } \mathbf{M}_i = \begin{bmatrix} p_n & & 0 & q_m & & 0 \\ \vdots & \ddots & & \vdots & \ddots & \\ p_{n-m+i+1} & \cdots & p_n & & & \\ & & & q_{m-n+i+1} & \cdots & q_m \\ \vdots & & \vdots & \vdots & & \vdots \\ p_{2i+2-m} & \cdots & p_{i+1} & q_{2i+2-n} & \cdots & q_{i+1} \\ X^{m-i-1}P & \cdots & \cdots & P & X^{n-i-1}Q & \cdots Q \end{bmatrix}$$

On verra au chapitre 3, dans la section 3.2.5, que les premières lignes de cette matrice, qui ne comportent que des coefficients, forment un bloc extrait de la matrice de Sylvester de P et Q.

Par définition du déterminant polynomial, le i-ème polynôme sous-résultant $S_i(P,Q)$ est de degré au plus i.

Le coefficient de degré i du polynôme $S_i(P,Q)$, qui peut être nul, est appelé i-ème *coefficient sous-résultant* de P et Q. Il est noté $sr_i(P,Q)$. Sa valeur est déterminée par les contributions des monômes de plus haut degré sur la dernière ligne de la matrice $\mathbf{M}_i$.

$$sr_i(P,Q) = det\ \mathbf{m}_i \text{ avec } \mathbf{m}_i = \begin{bmatrix} p_n & & 0 & q_m & & 0 \\ \vdots & \ddots & & \vdots & \ddots & \\ p_{n-m+i+1} & \cdots & p_n & & & \\ & & & q_{m-n+i+1} & \cdots & q_m \\ \vdots & & \vdots & \vdots & & \vdots \\ p_{2i+2-m} & \cdots & p_{i+1} & q_{2i+2-n} & \cdots & q_{i+1} \\ p_{2i+1-m} & \cdots & p_i & q_{2i+1-n} & \cdots & q_i \end{bmatrix}$$

L'intérêt des polynômes sous-résultants provient de leur lien direct avec les chaînes de pseudo-restes issues d'un couple de polynômes.

Considérons une chaîne de pseudo restes $F_1, \ldots, F_k$ issue des polynômes $P, Q \in D[X]$ tels que $deg(P) > deg(Q)$. Notons $P = p_0 + \cdots + p_n X^n$ et $Q = q_0 + \cdots + q_m X^m$. Rappelons les notations de la section 2.2.4, selon lesquelles $F_1 = P, F_2 = Q,\ deg(F_i) := n_i$ et pour $i >= 3$:

$$\beta_i F_i = \alpha_i F_{i-2} - Q_i F_{i-1} deg(F_i) < deg(F_{i-1}) (1)$$

Une récurrence sur i permet d'obtenir facilement pour chaque i une relation à la Bezout qui assure l'existence de U_i et V_i deux polynômes de $D[X]$ et d'un coefficient $\gamma_i \in D$ tels que :

$$U_i F_1 + V_i F_2 = \gamma_i F_i deg(U_i) <= m - ni - 1, deg(V_i) <= n - n_i - 1 (2)$$

Si γ_i est fixé, les polynômes U_i et V_i sont uniques.

Considérons maintenant le problème inverse qui consiste, étant donnée une PRS et une suite des coefficients γ_i, à déterminer pour chaque i les polynômes U_i et V_i qui réalisent l'identité (2).

On peut exprimer ce problème comme un système inhomogène d'équations linéaires dont les $n+m-2n_{i-1}+2$ inconnues sont les coefficients de U_i et V_i dans la base canonique. Chacune de ces $n+m-n_{i-1}+1$ équations est obtenue en égalisant les coefficients des monômes de même degré dans la relation (2).

Plus précisément, le second membre des $n+m-ni-1-n_i$ premières équations (issues des monômes de plus haut degré) est nul, tandis que celui des n_i+1 dernières (issues de monômes dont les coefficients sont potentiellement non nuls dans le polynôme F_i).

On dit que la chaîne de pseudo-restes est non défective si elle est la plus longue possibles, c'est à dire que les degrés ne chutent que de un à chaque étape de division, où encore que $deg(F_i) = n_i = m-i+2$ pour $i = 2...m$. Le cas non défectif est celui où on construit une chaîne de pseudo-restes à partir de deux polynômes qui sont premiers entre eux.

Dans ce cas, pour tout $i = 2...m, ni = k+1$ avec $k = m-i+1$ et dans le système linéaire décrit ci-dessus, on a $n+m-2k-1$ équations homogènes dont la matrice est :

$$\begin{bmatrix} p_n & & 0 & q_m & & 0 \\ \vdots & \ddots & & \vdots & \ddots & \\ p_{n-m+k+1} & \cdots & p_n & & & \\ & & & q_{m-n+k+1} & \cdots & q_m \\ \vdots & & \vdots & \vdots & & \vdots \\ p_{2k+2-m} & \cdots & p_{k+1} & q_{2k+2-n} & \cdots & q_{k+1} \end{bmatrix}$$

Quant aux équations inhomogènes, on peut les rassembler sous la forme d'une seule équation linéaire à coefficient polynomiaux dont le second membre est $\gamma_i F_i$ et la matrice est le vecteur ligne :

$$\begin{bmatrix} X^{m-k-1}P & \ldots & XP & P & X^{n-k-1}Q & \ldots & XQ & Q \end{bmatrix}$$

Finalement le système complet peut se représenter de la façon suivante :

$$\begin{bmatrix} p_n & & 0 & q_m & & 0 \\ \vdots & \ddots & & \vdots & \ddots & \\ p_{n-m+k+1} & \cdots & p_n & & & \\ & & & q_{m-n+k+1} & \cdots & q_m \\ \vdots & & \vdots & \vdots & & \vdots \\ p_{2k+2-m} & \cdots & p_{k+1} & q_{2k+2-n} & \cdots & q_{k+1} \\ X^{m-k-1}P & \ldots & P & X^{n-k-1}Q & \ldots & Q \end{bmatrix} \times W = \begin{bmatrix} 0 \\ \\ \vdots \\ \vdots \\ \\ 0 \\ \gamma_i F_i \end{bmatrix}$$

où W est le vecteur colonne $(u_{m-k-1}, \ldots, u_0, v_{n-k-1}, \ldots, v_0)$, concaténé des coordonnées de U_i et V_i dans la base canonique.

Le déterminant de ce système est non nul puisqu'il existe une solution unique au système. Ce déterminant ne dépend pas de γ_i : toute les chaînes de pseudo-restes issues des deux polynômes P et Q ont la même suite de degrés.

L'étude des polynômes sous-résultants est l'étude systématique de ces déterminants, sans connaître a priori le caractère défectif où non des chaînes de pseudo-restes issues des polynômes de départ.

De fait la matrice ci-dessus coïncide avec celle qui a permis la définition par déterminants des polynômes sous-résultants (suite de la définition 2.4.1).

Nous allons maintenant montrer la propriété fondamentale (voir le théorème de structure de la section 2.4.4) qui montre que la suite des polynômes sous-résultants de P et Q peut elle-même être obtenue comme une chaîne de pseudo-restes issue de P et Q.

2.4.2 Lemme de décalage

La propriété suivante, qui montre que $S_j(P,Q)$ est un multiple de $S_j(Q, preste(P,Q))$, constitue le noeud central de la preuve du théorème de structure des sous-résultants.

Lemme 2.4.1 (Lemme de décalage) *Soit F, G, H, B des polynômes non nuls de $D[X]$, de degrés respectifs $\phi, \gamma, \eta, \beta$, tels que :*

$$F + BG = H \text{ avec } \phi >= \gamma > \eta \text{ et } \beta = \phi - \gamma$$

Alors,

$$\begin{array}{lrcll}
(i) & S_j(F,G) & = & (-1)^{(\phi-j)(\gamma-j)} g_\gamma^{\phi-\eta} S_j(G,H) & 0 <= j < \eta \\
(ii) & S_\eta(F,G) & = & (-1)^{(\phi-\eta)(\gamma-\eta)} g_\gamma^{(\phi-\eta)} h_\eta^{\gamma-\eta-1} H & \\
(iii) & S_j(F,G) & = & 0 & \eta < j < \gamma - 1 \\
(iv) & S_{\gamma-1}(F,G) & = & (-1)^{\phi-\gamma+1} g_{\phi-\gamma+1} H &
\end{array}$$

Preuve On rappelle que par définition, pour $j < \gamma$:

$$S_j(F,G) = pdet(X^{\gamma-j-1}F, \ldots, XF, F, G, XG, \ldots, X^{\phi-j-1}G)$$

Dans le membre de gauche, remplacer chaque occurrence de F par H revient à ajouter à chacun des $\gamma - j$ premiers arguments une combinaison linéaire des $\phi - j$ derniers arguments.

En effet, pour tout $k \leq \gamma - j - 1$,

$$X^k F + X^k BG = X^k H \Rightarrow X^k F = X^k H - \sum_{i=k}^{\beta+k} b_i X^i G$$

où l'on note $B := b_0 + \cdots + b_\beta X^\beta$. Or comme $\gamma = \phi - \beta$, on a $\beta + k <= \phi - j - 1$ et tous les termes de la somme ci-dessus sont bien colinéaires aux $\phi - j$ derniers arguments.

Le déterminant polynomial étant multilinéaire et alterné, cette opération de remplacement ne change pas la valeur de *pdet*.

On remarque que les opérations sur les colonnes de la matrice $\mathbf{M}_i$, dont $S_j(F,G)$ est le déterminant, qui correspondent au remplacement de F par H simulent exactement les étapes élémentaires de la pseudo-division de F par H (qui est ici une vraie division euclidienne).

On obtient à la suite de cette opération de remplacement :

$$S_j(F,G) = pdet(X^{\gamma-j-1}H, \ldots, XH, H, G, XG, \ldots, X^{\phi-j-1}G)$$

Revenons à présent à la représentation matricielle de *pdet*. On note $G := g_0 + \cdots + g_\gamma X^\gamma$ et $H := h_0 + \ldots h_\eta X^\eta$ et on échange les deux blocs contenant respectivement les coefficients de H et ceux de G, ce qui introduit une puissance de (-1).

$$S_j(F,G) = (-1)^{(\phi-j)(\gamma-j)} det \begin{bmatrix} g_\gamma & & 0 & h_\phi & & 0 \\ \vdots & & & \vdots & \ddots & \\ \vdots & \ddots & \vdots & \vdots & & h_\phi \\ \vdots & & & \vdots & & \vdots \\ g_{\gamma-\phi+j+1} & \ldots & g_\gamma & h_{j+1} & \ldots & h_\gamma. \\ \vdots & & & \vdots & & \vdots \\ g_{2i+2-\phi} & \ldots & g_{j+1} & h_{2j+2-\gamma} & \ldots & h_{j+1} \\ X^{\phi-i-1}G & \ldots & G & X^{\gamma-j-1}H & \ldots & H \end{bmatrix}$$

Si $j >= \eta$, alors la matrice est triangulaire, et

$$S_j(F,G) = (-1)^(\phi - \eta)(\gamma - \eta) g_\gamma^{\phi-\eta} h_\eta^{\gamma-\eta-1} H for \eta <= j <= \gamma - 1$$

Ceci établit $(ii) - (iv)$. Maintenant si $j < \eta$, le déterminant est de la forme :

$$S_j(F,G) = (-1)^{(\phi-j)(\gamma-j)} det \begin{bmatrix} A & 0 \\ B & S_j(G,H) \end{bmatrix}$$

où A est un bloc carré triangulaire inférieur de taille $\phi - \eta$ dont tous les éléments sur la diagonale sont égaux à g_ϕ, ce qui prouve (i). □

Un argument élémentaire mais fondamental utilisé par la preuve de ce lemme est le fait qu'un polynôme est une sommes de monômes coefficientés.

Cette propriété n'est pas immédiate du point de vue de la représentation en forme de Horner que nous avons choisie. Pour établir ce résultat, nous avons donc dû montrer un théorème de changement de représentation, qui, une fois établi, permet de prouver les énoncés faisant intervenir la multilinéarité du déterminant.

2.4.3 Équivalence de représentations des polynômes en Coq

Pour obtenir la représentation des polynômes en formes de Horner comme somme de monômes coefficientés, il nous faut définir un monôme, c'est à dire un polynômes qui est une puissance positive ou nulle de l'indéterminée :

```
Definition mon(n:N):=
 match n with
  |N0 ⇒ P1
  |Npos i ⇒ PX (Pc c1) i c0
 end.
```

Code 2.10: *Monôme en forme de Horner creuse*

Il faut aussi définir de façon abstraite une somme de tels monômes pondérés par des coefficients. En fait on définit plutôt la somme finie des premiers termes d'une suite infinie de polynômes. Cette formalisation recouvre en particulier le cas où les termes de la suite infinie sont les monômes coefficientés.

```
Fixpoint Psum(f:nat  → Pol)(N:nat){struct N}:Pol :=
match N with
 |O ⇒ f O
 |S N ⇒ (Psum f N) + (f (S N))
end.
```

Code 2.11: *Somme finie des premiers termes d'une suite infinie de polynômes*

Si on se donne un polynôme `P`, la fonction qui à un entier `i` associe le monôme X^i multiplié par le coefficient de `P` sur X^i s'écrit :

```
(fun i ) scal (get_coef i P) (mon i))
```

La somme des monômes coefficientés d'un polynôme est donc définie par :

```
Definition sum_of_Pol(P:Pol) :=
Psum (fun i ) scal (get_coef i P) (mon i) (nat_deg P).
```

Code 2.12: *Somme des monômes coefficientés d'un polynôme*

où `nat_deg` est le degré d'un polynôme en forme normale de Horner creuse, c'est à dire la somme cumulée des exposants imbriqués dans les constructeurs `PX` du polynôme.

On note `=p` la relation d'égalité sur les polynômes. Le théorème de changement de représentation s'énonce :

```
Theorem P_mon_sum : ∀ P, P =p sum_of_Pol P.
```

Code 2.13: *Changement de représentation*

Ce théorème se prouve par induction sur la structure du polynôme P et utilise les propriétés de la construction `Psum`.

Par exemple,`Psum` est compatible avec l'égalité extentionnelle des suites de polynômes :

```
Lemma Psum_ext : ∀ N f g,(∀ n, f n != g n) )
                                    (Psum f N) =p (Psum g N).
```

Code 2.14: `Psum` *est compatible avec l'égalité extentionnelle*

2.4.4 Théorème de structure des polynômes sous-résultants

Dans la section 2.4.2, on s'est intéressé au comportement des polynômes sous-résultants lorsqu'on les combine avec une division euclidienne des deux polynômes de départ.

L'itération du lemme 2.4.1 permet d'établir la relation entre chaque polynôme sous-résultant et un des éléments de chaîne de pseudo-restes. Comme on utilise la multilinéarité du déterminant polynomial, on obtient une relation de similarité.

Le lemme technique suivant est un corollaire direct du lemme 2.4.1.

Lemme 2.4.2 *Soit une chaîne de pseudo-restes $F_1, \ldots, F_k$, définie par les relations euclidiennes, pour $i = 3 \ldots k$:*

$$\beta_i F_i = \alpha_i F_{i-2} - Q_i F_{i-1} \quad deg(F_i) < deg(F_{i-1}) (1)$$

On note n_i le degré de F_i et c_i le coefficient dominant de F_i pour $i = 3 \ldots k$.

Pour $i = 3 \ldots k$,

(i) Pour $0 <= j < n_i$,

$$S_j(F_{i-2}, F_{i-1})\alpha_i^{i-1-j} = S_j(F_i - 1, F_i)\beta_i^{n_{i-1}-j} c_i^{n_i-1-n_i} - 1(-1)^{(n_{i-2-j})(n_{i-1-j})}$$

(ii) $S_{n_i}(F_{i-2}, F_{i-1})\alpha_i^{n_{i-1}-n_i} = F_i \beta_i^{n_{i-1}-n_i} c_i^{n_{i-1}-n_i-1} c_{i-1}^{n_{i-2}-n_i} (-1)^{(n_{i-2}-n_i)(n_{i-1}-n_i)}$

(iii) $S_j(F_{i-2}, F_{i-1}) = 0$ *pour* $n_i < j < n_i - 1 - 1$

(iv) $S_{n_{i-1}-1}(F_{i-2}, F_{i-1})\alpha_i = F_i \beta_i c_{i-1}^{n_{i-2}-n_{i-1}+1} (-1)^{n_{i-2}-n_{i-1}+1}$

Preuve Il s'agit exactement du lemme 2.4.1, appliqué à $F = \alpha_i F_{i-2}, G = F_{i-1}$ et $H = \alpha_i F_i, B = -Q_i$ en utilisant la propriété de multilinéarité :

$$S_j(aF, bG) = a^{\gamma-j} b^{\phi-j} S_j(F, G)$$

□

C'est ce lemme technique qui permet de montrer directement le théorème de structure des polynômes sous-résultants. Dans la section 2.4.1, on a vu que les polynômes sous-résultants apparaissent dans les déterminants des systèmes qui définissent les coefficients de Bezout des relations euclidiennes dans une chaîne de pseudo-restes.

Le théorème de structure montre qu'outre ce lien entre polynômes sous-résultants et chaînes de pseudo-restes, la suite des polynômes sous-résultants eux-même peut être obtenue comme une certaine chaîne de pseudo-restes dont on va préciser les coefficients.

Pour plus de clarté et pour mettre en valeur les polynômes similaires, on ne donne les valeurs exactes des coefficients de similarité que dans la preuve.

Théorème 2.4.3 (Théorème fondamental) *Avec les mêmes notations que dans le lemme 2.4.1 ci-dessus,*

(i) $S_j(F_1, F_2) = 0$, *pour* $0 <= j < n_k$

(ii) $S_j(F_1, F_2) = 0$ *pour* $n_i < j < n_{i-1} - 1$

(iii) $S_{n_i}(F_1, F_2)$ *et* F_i *sont similaires.*

(iv) $S_{n_i-1-1}(F_1, F_2)$ *et* F_i *sont similaires, pour* $i = 3, \ldots k$.

Preuve Pour $0 \leq j < n_{i-1}$, l'itération du lemme 2.4.2-(i) jusqu'à $3 <= i \leq k+1$ établit que :

$$(*) \quad S_j(F_1, F_2), S_j(F_{i-2}, F_{i-1})$$

Lorsque $i = k + 1$, la relation $(*)$ s'écrit :

$$S_j(F_1, F_2) \sim S_j(F_{k-1}, F_k)$$

Pour $0 \leq j < n_k$, le lemme 2.4.1 assure que $S_j(F_{k-1}, F_k) = 0$ d'où (i).

Lorsque $n_i < j < n_{i-1} - 1$, le lemme 2.4.2-(iii) combiné avec $(*)$ montre (ii).

On peut affiner la relation $(*)$ et obtenir par récurrence sur i la valeur des coefficients de similarité :

$(**)$ $S_j(F_1, F_2) \prod_{l=3}^{i-1} \alpha_l^{n_{l-1}-j} =$

$$S_j(F_{i-1}, F_{i-2}) \prod_{l=3}^{i-1} [\beta_l^{n_{l-1}-j} c_{l-1}^{n_{l-2}-n_l} (-1)^{(n_l-2-j)(n_l-1-j)}]$$

Les formules exactes que nous allons établir pour (iii) et (iv) sont respectivement :

(ii) : $S_{n_i}(F_1, F_2) \prod_{l=3}^{i} \alpha_i^{n_{l-1}-n_i} =$

$$F_i c_i^{n_{i-1}-n_i-1} \prod_{l=3}^{i} [\beta_l^{n_{l-1}-n_i} c_{l-1}^{n_l-2-n_l} (-1)^{(n_l-2-n_i)(n_l-1-n_i)}]$$

et :

(iv) : $S_{n_{i-1}-1}(F_1, F_2) \prod_{l=3}^{i} \alpha_l^{n_{l-1}-n_{i-1}+1} =$

$$F_i c_{i-1}^{1-n_i-1+n_i} \prod_{l=3}^{i} [\beta_l^{n_{l-1}-n_{i-1}+1} c_{l-1}^{n_{l-2}-n_l} (-1)^{(n_{l-2}-n_{i-1}+1)(n_{l-1}-n_{i-1}+1)}]$$

Soit i tel que $0 \leq i <= k$. Pour $j = n_i$, le lemme 2.4.2-(ii) combiné avec $(**)$ montre (iii).

Pour finir, le lemme 2.4.2-(iv) et $(**)$ prouvent (iv). □

2.4.5 Algorithme des sous-résultants, calcul de pgcd

Le théorème fondamental 2.4.3 montre que les éléments non nuls d'une suite de polynômes sous-résultants sont similaires aux polynômes d'une chaîne de pseudo-restes quelconque issue des mêmes polynômes de départ.

En fait le théorème 2.4.3 permet de connaître exactement la structure de la suite des polynômes sous-résultants issus de deux éléments P et Q de $D[X]$. On pourra se reporter au théorème de structure des sous-résultants : voir Théorème 8.30 pp.307 et suivantes dans (Basu, Pollack, et Roy 2006)

En particulier cette suite a la propriété suivante : deux polynômes séparés par des zéros consécutifs dans la suite des sous-résultants sont similaires. Si l'on efface de la suite des sous-résultants les polynômes nuls et les "redondances ", c'est à dire un élément de chaque paire de polynômes similaires, on obtient une suite dans laquelle les éléments sont deux à deux similaires à toute chaîne de pseudo-restes issue de P et Q.

On peut même, en choisissant de façon appropriée les coefficients α_j et β_j des relations euclidienne, construire une chaîne de pseudo-restes dont les éléments seront *exactement* les éléments non nuls de la suite des polynômes sous-résultants où l'on a supprimé les redondances.

Une telle chaîne de pseudo-restes est appelée *chaîne des sous-résultants.*

La difficulté du problème de calcul de pgcd par chaîne de pseudo-restes est de trouver des valeurs raisonnables de coefficients qui permettent d'effectuer les divisions euclidiennes en restant dans l'anneau. Par valeurs raisonnables, on entend sans tomber dans un excès (la condition suffisantes mais souvent trop grossière du facteur de Jacobi), ou dans un autre (le calcul récursif des contenus).

Du fait que la définition même des polynômes sous-résultants (voir définition 2.4.1) assure leur appartenance à $D[X]$, lorsqu'on utilise les coefficients α_j et β_j qui permettent d'obtenir la chaîne des sous-résultants, on est assuré par avance que le reste de la division restera dans l'anneau.

On présente sur la figure 2.2 l'algorithme qui calcule la suite des polynômes sous-résultants, en utilisant la chaîne des sous-résultants et en intercalant les zéros et les

polynômes similaires.

La preuve de correction de cet algorithme n'a pas encore été formalisée en Coq. Pour ce faire il faut :

- montrer que les quantités $(-1)^{\frac{(j-k-1)(j-k)}{2}} \frac{t_{j-1}^{j-k}}{s_j^{j-k-1}}$ restent dans l'anneau D : voir Proposition 8.42 p.314 dans (Basu, Pollack, et Roy 2006) ;
- instancier le théorème fondamental avec les valeurs des coefficients de similarité ci-dessus pour montrer que les restes sont bien proportionnels aux polynômes sous-résultants, donc des éléments de $D[X]$.

2.5 Conclusion

Lorsque l'on travaille avec des polynômes à coefficients dans un anneau, l'algorithme d'Euclide ne peut plus être utilisé pour calculer un pgcd polynomial.

Nénmoins, si l'on ne peut pas toujours diviser un polynôme P arbitrairement choisi par un autre polynôme Q, il est toujours possible d'effectuer une division euclidienne d'un multiple de P par une puissance suffisante du coefficient dominant de Q.

En utilisant cette pseudo-division euclidienne, on peut définir une généralisation de l'algorithme d'Euclide pour le cas des coefficients dans des anneaux. Une approche naïve de ces chaînes de pseudo-restes conduit néanmoins à une croissance exponentielle des coefficients des polynômes.

L'algorithme le plus efficace qui soit basé sur cette idée est celui des chaînes de sous-résultants. Les polynômes sous-résultants sont définis comme les déterminants polynomiaux de certaines familles de polynômes, en fait des produits des polynômes P et Q par des monômes. Ils sont donc par définition dans l'anneau des polynômes de départ, sans coefficient fractionnaire.

On montre ensuite que ces polynômes peuvent être obtenus comme une chaîne de pseudo-restes, et que la croissance de leur coefficients est raisonnable, c'est à dire quadratique.

Signalons que l'on peut encore optimiser l'algorithme des sous-résultants (Ducos 2000), pour obtenir par une chaîne de pseudo-restes des polynômes auxquels les polynômes sous-résultants de la définition 2.4.1 sont *proportionnels*, c'est à dire qu'on diminue encore la taille des coefficients.

Dans ce travail j'ai [roposé un algorithme de division euclidienne pour les polynômes en forme de Horner creuse (voir le chapitre 1) qui repose sur une récurrence structurelle sur le dividende.

Nous avons programmé l'algorithme des sous-résultants 2.2 en utilisant cette procédure de division. Ce programme est utilisé intensivement pour les nombreux calculs de pgcd qui interviennent dans l'algorithme de CAD, ainsi que lors de la phase d'élimination (voir le chapitre 3).

Pour prouver formellement cet algorithme, nous proposons une fomalisation des déterminants polynomiaux, qui repose sur une définition en Coq des déterminants, qui est à ma connaissance la première formalisation de ce type dans le système.

Enfin nous avons achevé la preuve formelle du lemme de décalage, qui est le pilier de la preuve du théorème fondamental 2.4.3 des sous-résultants. Pour obtenir une preuve

complète de ce dernier théorème, il à reste à prouver des identités algébriques d'anneau dans lesquelles la difficulité sera probablement de trouver une solution pratique pour définir et manipuler aisément les symboles $\prod$ et $\sum$, c'est à dire des produits et sommes d'un nombre variable de termes.

À notre connaissance, les seules expériences de calcul de pgcd de polynômes certifiés avec un assistant à la preuve portent sur des calculs de polynômes à coefficients dans un corps algébriquement clos. En particulier, ce peut être un ingrédient de procédures de décision ou de semi-décision pour les corps algébriquement clos.

D. Delahaye et M. Mayero ont utilisé un interfaçage du système Coq avec le système de calcul formel Maple pour écrire une telle procédure de décision (Delahaye et Mayero 2005). Les calculs des pgcd de polynômes sont laissés au système de calcul formel. Les identités d'anneau qui prouvent les spécifications des calculs effectués par Maple sont vérifiées en Coq. Malheureusement le code de cette procéure n'est pas encore disponible. Il sera intéressant de comparer les performances respectives de notre approche complètement autarcique et de leur utilisation de la puisance de calcul du logiciel Maple.

Notations :
Soit $P = p_0 + \cdots + p_n X^n$ et $Q = q_0 + \cdots + q_m X^m$ deux éléments de $D[X]$.
On note S_j le j-ème polynôme sous-résultant de P et Q.
On note s_j le j-ème coefficient sous-résultant de P et Q : c´est le coefficient de S_j sur X^k
On note t_j le coefficient dominant de S_j.

Initialisation :

$S_n := P$	$S_{n-1} := Q$	$S_m := q_m^{n-m-1} Q$	$S_l := 0$ pour l de $m+1$ à $n-2$
$s_n := 1$	$s_{n-1} := q_m$	$s_m := q_m^{n-m}$	$s_l := 0$

$i := n+1, j := n$

Procédure :
- Tant que $S_{j-1} \neq 0$, soit $k := deg(S_{j-1})$
  - Si $k = j-1$
    - $s_{j-1} := t_{j-1}$
    - $S_{k-1} := -\dfrac{Reste(s_{j-1}^2 S_{i-1}, S_{j-1})}{s_j t_{i-1}}$
  - Sinon, $k < j-1$
    - $s_{j-1} := 0$
    - $s_k := (-1)^{\frac{(j-k-1)(j-k)}{2}} \dfrac{t_{j-1}^{j-k}}{s_j^{j-k-1}}$
    - $S_k := \dfrac{s_k S_{j-1}}{t_{j-1}}$
    - $S_l := 0$ pour l de $j-2$ à $k+1$
    - $s_l := 0$ pour l de $j-2$ à $k+1$
    - $S_{k-1} := -\dfrac{t_{j-1} s_k S_{i-1}, S_{j-1}}{s_j t_{i-1}}$
    - $t_{k-1} := coefdom(S_{k-1})$
    - $i := j,\ j := k$
- Pour $l = 0 \ldots j-2$
  - $S_l := 0$
  - $s_l := 0$
- Retourner $\mathbf{S} := S_n, \ldots, S_0$ et $\mathbf{s} := s_n, \ldots, s_0$

FIG. 2.2: *Algorithme des sous-résultants*

Chapitre 3

Le problème de décision dans la théorie des réels

Dans ce chapitre nous décrivons une méthode de décision pour la théorie du premier ordre des nombres réels. Elle constitue la base d'une procédure de génération automatique de preuves pour les énoncés du premier ordre sur les réels en Coq, c'est à dire d'une tactique Coq. Depuis la preuve historique de décidabilité établie par Tarski, dans les années trente et publiée pour la première fois dans les années cinquante (Tarski 1951), de nombreuses améliorations ont été proposées pour améliorer la complexité de la méthode.

3.1 Décidabilité dans les corps réels clos

La théorie qui nous intéresse ici est celle que Tarski appelle dans son article historique, *algèbre et géométrie élémentaire des nombres réels*. Il s'agit en fait de la théorie du premier ordre des corps réels clos ordonnés, ou plus informellement, des énoncés mathématiques qui expriment des conditions de signe sur les valeurs prises en des point réels par des polynômes à plusieurs variables et à coefficients rationnels.

Le genre de problème dont on saura décider automatiquement la validité avec une telle procédure sera par exemple :

- Tout polynôme de degré un a une racine (réelle) ;
- Tout polynôme de degré dix-huit au moins une racine (réelle) ;
- Deux ellipses ont au plus quatre points d'intersection
- Un polynôme de degré deux a au plus deux racines (réelles)
- Il existe un polynôme de degré deux qui n'a qu'une seule racine
- $\forall \epsilon, \exists \delta, |x| < \delta \Rightarrow -\epsilon \leq x^5 \leq \epsilon$
- ...

Par contre, bien sûr, on ne capture pas les énoncés "d'ordre supérieur" de la géométrie ou de l'algèbre. Par exemple, on ne saura pas prouver ou infirmer :

"Tout polynôme a au moins une racine (réelle)"

De la même façon, on ne saura pas, en général, donner plus d'information sur les points réels que l'on construit. Si tel était le cas, on pourrait, par exemple, tirer de cette procédure un algorithme décidant si une équation diophantienne a une solution ou non, or ce *dixième problème de Hilbert*, est indécidable (Matiyasevic 1970).

Même si historiquement les géomètres n'ont pas relié la publication de Tarski à leur intérêts en géométrie algébrique (à la notable exception de Seidenberg), cette propriété de décidabilité est indissociable de la géométrie *semi-algébrique réelle* qui étudie les variétés réelles définies par des équations et inéquations polynomiales.

3.1.1 Corps réels clos

Définition 3.1.1 (Corps réel clos) *Les axiomes suivants définissent la théorie des corps réels clos intuitionnistes dans le langage $\mathcal{L} = \{+, \cdot, 0, 1, P\}$, où P est un prédicat unaire (interprété comme un prédicat de positivité).*

1. *Axiomes définissant un anneau commutatif*
2. *P(1)*
3. $P(x) \vee P(-x) \Rightarrow x$ *est inversible*
4. $\neg P(0)$
5. $\neg P(-x) \wedge \neg P(x) \Rightarrow x = 0$
6. $P(x) \wedge P(y) \Rightarrow P(x+y) \wedge P(x.y)$
7. $P(x+y) \Rightarrow P(x) \vee P(y)$

Dans la suite $x < y$ représentera $P(y-x)$ et $x \leq y$ représentera $\neg P(x-y)$. Nous introduisons également le symbole $>$, tel que $x > y$ représente $-x < -y$.

8. Schéma d'axiome pour le théorème des valeurs intermédiaires pour les polynômes $f(x) = y_0 x^n + y_1 x^{n-1} \ldots y_n$

$$f(u) < 0 < f(v) \wedge u < v \Rightarrow \exists x, u < x < v \wedge f(x) = 0$$

Nous étendons cette théorie avec l'axiome de décidabilité du prédicat de positivité, pour constituer la définition de la théorie des corps réels clos discrets.

9. $P(x) \vee \neg P(x)$

E. Palmgren a montré (Palmgren 2002) que l'ajout de ce dernier axiome conduit à la théorie classique des corps réels clos :

Théorème 3.1.1 *La théorie des corps réels clos est équivalente en logique intuitionniste à la théorie classique des corps réels clos (ceux pour lesquels le principe du tiers-exclu vaut pour toute formule).*

Les nombres réels constructifs fournissent un modèle pour la théorie des corps réels clos mais bien sûr pas pour la théorie des corps réels clos discrets.

Les notions de *corps réel*, et de *corps réel clos* sont introduite en 1927 dans (Artin et Schreier 1927). Une structure de corps réel est un corps ordonné : les opérations de corps doivent être compatibles avec une relation qui ordonne totalement le corps. Historiquement, Artin et Schreier ont donné une caractérisation algébrique de cette propriété en définissant un corps réel comme un corps dans lequel -1 ne peut être représenté comme une somme de carrés. De façon équivalente, la nullité d'une somme de carrés entraîne la nullité de chaque carré.

Un *corps réels clos* est défini algébriquement (Artin et Schreier 1927) comme un corps réel n'admettant pas d'extension algébrique réelle. On peut maintenant vérifier que l'on a

ainsi décrit les propriétés qui font la spécificité de l'algèbre réelle. Pour commencer, tout corps réel clos ne peut être ordonné que d'une et une seule manière, grâce au prédicat de positivité[1]. On vérifie également que tout polynôme de degré impair possède au moins une racine. Un corps réel clos n'est pas algébriquement clos mais le devient lorsqu'on lui ajoute une racine de X^2+1 (souvent notée i...).

Le corps des nombres réels $\mathbb{R}$ est un corps réel clos, ainsi que celui des nombres algébriques réels. Il existe d'autres exemples de tels corps, en particulier des corps réels clos non archimédiens, comme l'ensemble des germes des fonctions semi-algébriques continues à droite de 0.

3.1.2 Élimination des quantificateurs

Soit R un corps réel clos et D un sous-anneau de R. Le *langage des corps ordonnés avec coefficients dans* D est défini simultanément avec l'ensemble des *variables libres* d'une formule :

Définition 3.1.2 (Langage des corps ordonnés avec coefficients dans D)
- *Un* atome $P = 0$ *ou* $P > 0$ *ou* $P < 0$, *où* P *est un polynôme, élément de* $D[X_1, \ldots, X_n]$ *est une formule* Φ *dont les variables libres sont* $Libres(\Phi) = \{X_1, \ldots, X_n\}$.
- *Si* Φ_1 *et* Φ_2 *sont des formules, alors* $\Phi_1 \wedge \Phi_2$ *et* $\Phi_1 \vee \Phi_2$ *sont des formules avec* $Libres(\Phi_1 \wedge \Phi_2) = Libres(\Phi_1 \vee \Phi_2) = Libres(\Phi_1) \cup Libres(\Phi_2)$
- *Si* Φ *est une formule, alors* $\neg\Phi$ *est une formule et* $Libres(\neg\Phi) = Libres(\Phi)$
- *Si* Φ *est une formule et* $X \in Libres(\Phi)$ *alors,* $\exists X\Phi$ *et* $\forall X\Phi$ *sont des formules et* $Libres(\exists x\Phi) = Libres(\forall x\Phi) = Libres(\Phi)\backslash\{X\}$

Lorsque D est $\mathbb{Z}$, l'anneau des entiers ou $\mathbb{Q}$ le corps des rationnels, on parlera de *formules de Tarski.*

Nous aurons aussi besoin dans la suite des deux définitions suivantes :

Définition 3.1.3 (Ensembles algébriques, ensembles semi-algébriques) *Soit* $\mathcal{P}$ *un sous-ensemble fini de* $R[X_1, \ldots, X_n]$. *L'ensemble des zéros de* $\mathcal{P}$ *est défini par* $Z(\mathcal{P}) = \{x \in R^n | \bigwedge_{P\in\mathcal{P}} P(x) = 0\}$. *Un tel ensemble de* R^n *est appelé un ensemble* algébrique *de* R^n. *Les ensembles* semi-algébriques *de* R^n *sont la plus petite famille contenant les ensembles algébriques et les ensembles définis pas des inégalités polynomiales, qui soit close par complémentaire, union finie et intersection finie.*

Si D *est un sous-anneau de* R *un ensemble semi-algébrique défini par des équations/inégalités de polynômes à coefficients dans* D *est dit* défini sur D.

Définition 3.1.4 (Fonctions semi-algébriques) *Soit* $S \subset R^k$ *et* $T \subset R^l$ *des ensembles semi-algébriques. Une fonction* $f : S \to T$ *est appelée une fonction* semi-algébrique *si son graphe est un sous-ensemble semi-algébrique de* R^{k+l}.

Une des propriétés essentielles des ensembles semi-algébriques est leur stabilité par projection :

[1] L'ordonnabilité des corps réels est longtemps resté un résultat non constructif, jusqu'aux travaux de Hollkott puis Lombardi et Roy.

Théorème 3.1.2 (Stabilité par par projection) *Soit R un corps réel clos, la projection sur R^k d'un ensemble semi-algébrique de R^{k+1} défini sur D est un ensemble semi-algébrique défini sur D.*

Le pendant logique de ce théorème de stabilité est connu sous le nom de principe de Tarski-Seidenberg :

Théorème 3.1.3 (Tarski-Seidenberg) *Soit R' un corps réel clos qui contient le corps réel clos R. Si Φ est une formule du langage des corps ordonnés à coefficients dans R, alors Φ est valide dans R si et seulement si elle est valide dans R'.*

Ces deux derniers théorèmes seront prouvés par les algorithmes de décisions dont la présentation suit.

La théorie des corps réels clos discrets admet l'élimination des quantificateurs :

Théorème 3.1.4 (Élimination des quantificateurs) *Soit $\Phi(Y)$ une formule dans le langage des corps ordonnés avec coefficients dans un anneau ordonné D contenu dans le corps réel clos R. Il existe une formule sans quantificateurs $\Psi(Y)$ avec coefficients dans D, telle que pour tout $y \in R^k$, $\Psi(y)$ est vraie si et seulement si $\Phi(y)$ est vraie.*

L'élimination des quantificateurs peut être vue comme une conséquence de la stabilité de la famille des ensembles semi-algébriques de R^k par projection. Voir par exemple (Basu, Pollack, et Roy 2003), pp 60 et suivantes.

Dans tout ce qui suit, on supposera que le corps réel clos étudié est $\mathbb{R}$, qui a la particularité d'être archimédien, même si des procédures de décisions sont également disponibles pour les structures non-archimédiennes.

3.1.3 Le problème de décision

On s'intéresse dans toute la suite au problème de décision des formules du premier ordre de de la théorie de $\mathbb{R}$ avec coefficients dans un anneau D, sur lequel les calculs sont exacts et le signe des éléments est calculable. Ce anneau sera $\mathbb{Q}$ ou $\mathbb{Z}$. La validité d'une telle formule est liée aux propriétés de la variété algébrique définie par les polynômes qui composent ses atomes.

Définition 3.1.5 ($\mathcal{P}$-formules) *Soit $\mathcal{P} \subset \mathbb{Q}[X_1, \ldots, X_n]$ une famille de polynômes. Une $\mathcal{P}$-formule est une formule du premier ordre de la théorie de $\mathbb{R}$ définie sur $\mathbb{Q}$ dont les atomes sont des $\mathcal{P}$-atomes, c'est à dire de la forme $P = 0$ ou $P < 0$ ou $P > 0$, avec $P \in \mathcal{P}$.*

Les différents algorithmes que nous allons présenter ont un but commun : étant donnée une famille finie de polynômes $\mathcal{P} \subset \mathbb{Q}[X_1, \ldots, X_n]$, on veut calculer une partition *adaptée* à la famille $\mathcal{P}$. On supposera dans toute la suite que l'on a fixé un ordre sur les indéterminées $X_1, \ldots X_n$.

Définition 3.1.6 (Partition adaptée) *Soit $\mathcal{P} \subset \mathbb{Q}[X_1, \ldots, X_n]$ une famille de polynômes. Une partition de l'espace $\mathbb{R}^n$ en un nombre* fini *de cellules est dite* adaptée *à la famille $\mathcal{P}$ si et seulement si sur toute cellule de la partition, le signe de chacun des éléments de $\mathcal{P}$ est constant.*

Nous allons montrer qu'il est toujours possible, étant donnée une famille de polynômes, de calculer une partition de l'espace qui lui soit adaptée, et de plus de calculer le signe de chacun des éléments de la famille sur chaque cellule.

L'énumération de ces cellules donne toutes les conditions de signes simultanément réalisables par les éléments de $\mathcal{P}$. Ceci permet déjà de décider tout problème existentiel.

Cependant cette simple énumération ne sera pas suffisante pour une procédure de décision complète. Considérons par exemple la formule

$$\exists x \forall y P(x, y) = 0$$

Les polynômes $P_1(X, Y) = XY$ et $P_2(X, Y) = X + Y$ possèdent la même liste de conditions de signes réalisables : $\{[-]; [0]; [+]\}$. Par contre, la formule est valide pour $P = P_1$ mais pas pour $P = P_2$.

Par contre, on saura résoudre ce problème si on sait calculer un point par composante connexe semi-algébrique de la variété définie par la famille de départ, ainsi que les signes réalisés par les polynômes en chacun des points de cet échantillon.

Il est néanmoins difficile en général de calculer *exactement* un point par composante connexe, et on calculera en général une partition comportant trop de cellules, et un point par cellule de cette partition.

Les algorithmes auxquels nous nous intéressons ici calculent des partitions adaptées particulières, appelées *décompositions algébriques cylindriques.*

Définition 3.1.7 (Décomposition Algébrique Cylindrique) *Une* décomposition algébrique cylindrique *(CAD) de $\mathbb{R}^k$ est une suite $\mathcal{S}_1, \ldots, \mathcal{S}_k$, telle que pour tout $1 \leq i \leq k$, $\mathcal{S}_k$ est une partition de $\mathbb{R}$ en sous-ensembles semi-algébriques, appelés* cellules de niveau *i. Ces cellules satisfont les propriétés suivantes :*

- *Toute cellule de niveau 1 est soit un intervalle, soit est réduite à un point ;*
- *Pour tout $1 \leq i < k$ et pour toute cellule S de niveau i, il existe un nombre fini l_S de fonctions semi-algébriques continues*

$$\xi_1 < \cdots < \xi_{l_S} : S \to \mathbb{R}$$

telles que le cylindre $S \times \mathbb{R} \subset \mathbb{R}^{i+1}$ est l'union disjointe de cellules de $\mathcal{S}_{i+1}$. Une cellule de $\mathcal{S}_{i+1}$ est constituée :

- *soit par le graphe de l'une des fonctions ξ_j pour $j = 1 \ldots l_i$:*

$$\{(x, y) \in S \times \mathbb{R} \mid y = \xi_j(x)\}$$

- *soit par une bande ouverte du cylindre, bordée par les graphes des fonctions ξ_j et ξ_{j+1} pour $j = 0 \ldots l_S + 1$ où par convention $\xi_0 = -\infty$ et $\xi_{l_S+1} = +\infty$:*

$$\{(x, y) \in S \times \mathbb{R} \mid \xi_j < y < \xi_{j+1}(x)\}$$

On dit qu'une décomposition algébrique cylindrique de $\mathbb{R}^k$ est adaptée à une famille de polynômes $\mathcal{P} \subset \mathbb{Q}[X_1, \ldots, X_k]$ si les cellules de niveau k forment une partition adaptée de $\mathbb{R}^k$.

Lorsqu'on a calculé une décomposition algébrique cylindrique adaptée à une famille de polynômes, on en extrait un type particulier d'échantillons exhaustifs de points, appelé échantillon cylindrique.

Définition 3.1.8 (Arbres) *Pour tout entier k on note $\mathbb{N}^k$ l'ensemble des mots finis de longueur exactement k sur $\mathbb{N}$. On note $\sqsubset_{pref}$ l'ordre préfixe sur les mots et $|u|$ la longueur d'un mot u.*

Un arbre *est un sous-ensemble de $\bigcup_{i=0}^{\infty} \mathbb{N}^i$ stable par préfixe.*

Pour tout entier k, un arbre $\mathcal{T}$ de profondeur k *est un sous ensemble de $\bigcup_{i=0}^{k} \mathbb{N}^i$ stable par préfixe, tel que :*

$$\forall u \in \mathcal{T}, \exists v \in \mathcal{T}\, u \sqsubset_{pref} v \text{ et } |v| = k$$

Un tel arbre $\mathcal{T}$ est dit à branchement fini *si pour tout élément $u \in \mathcal{T}$, u n'admet qu'un nombre fini de suffixes dans $\mathcal{T}$.*

Si $\mathcal{T}$ est un arbre de profondeur k, tout mot de longueur k appartenant à $\mathcal{T}$ est appelé une feuille *de $\mathcal{T}$ et tout mot de longueur strictement plus petite que k est appelé nœud interne de $\mathcal{T}$.*

Définition 3.1.9 (Échantillon cylindrique de points) *Soit $\mathcal{P} = \{P_1 \dots P_k\} \subset \mathbb{Q}[X_1, \dots, X_n]$ une famille finie de polynômes. On note $sign(\mathcal{P})$ la fonction qui à un point z de $\mathbb{R}^n$ associe la liste ordonnée des signes de $P_1(z) \dots P_k(z)$:*

$$sign(\mathcal{P}) : \mathbb{R}^n \to \{-, 0, +\}^k$$

Soit $\mathcal{T}$ un arbre à branchement fini de profondeur $n+1$. Soit s une fonction qui à tout feuille de $\mathcal{T}$ associe un élément de $\{-, 0, +\}^k$. Soit alg une fonction qui à tout noeud interne de $\mathcal{T}$ associe un nombre réel algébrique.

On dit que $\mathcal{T}$ est un échantillon cylindrique de points *pour la famille $\mathcal{P}$ si pour toute feuille $f = \overline{u_0 \dots u_{n+1}}$ de l'arbre $\mathcal{T}$:*

$$sign(\mathcal{P})(alg(\overline{u_0}), alg(\overline{u_0 u_1}), \dots, alg(\overline{u_0 \dots u_n})) = f(\overline{u_0 \dots u_{n+1}})$$

L'ensemble :

$$\{(alg(\overline{u_0}), alg(\overline{u_0 u_1}), \dots, alg(\overline{u_0 \dots u_n})) | \overline{u_0 \dots u_n} \in \mathcal{T}\} \subset \mathbb{R}^n$$

*définit l'*ensemble des points *de l'échantillon cylindrique.*

La figure 3.1 représente un exemple d'échantillon cylindrique de points issu d'une décomposition algébrique cylindrique de la famille de polynômes réduite au singleton XY.

Chaque noeud interne est donc étiqueté par un nombre réel, en fait un nombre algébrique, choisi comme représentant dans un intervalle. Dans la suite, par souci de clarté, dans les représentation d'échantillons cylindriques de points issus d'une CAD, on étiquettera un noeud interne non pas par la coordonnée du point mais par cet intervalle dont le point est un échantillon.

Par abus de langage, sauf mention du contraire, on appellera un échantillon cylindrique de points pour une famille de polynômes, un échantillon cylindrique de points exhaustif issu du calcul d'une CAD pour cette famille.

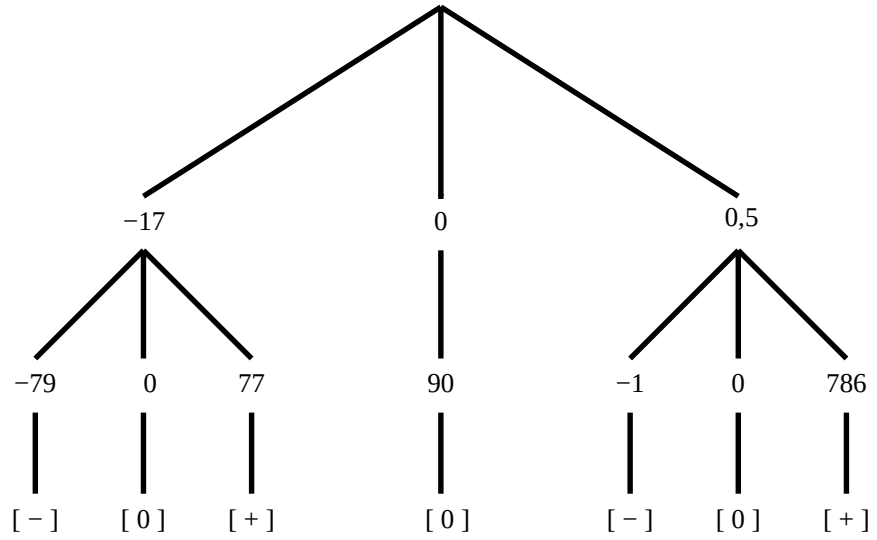


FIG. 3.1: *Échantillon cylindrique de points issu d'une CAD de la famille* $\{XY\}$

Le problème à une variable

Soit $\mathcal{P} = \{P_1, \ldots, P_s\} \subset \mathbb{Q}[X]$ une famille finie de polynômes à une variable. Une $\mathcal{P}$-formule est une combinaison booléenne de formules de la forme : $\forall x\ B(x)$ ou $\exists x\ B(x)$ dans laquelle $B(x)$ est une combinaison booléenne de $\mathcal{P}$-atomes.

Le calcul d'une décomposition algébrique cylindrique adaptée à $\mathcal{P}$ donne une liste de points $x_1 < \cdots < x_p$ telle que les signe de chacun des polynômes de la famille $\mathcal{P}$ a un signe constant sur les intervalles ouverts $]x_i, x_{i+1}[$.

Un échantillon cylindrique exhaustif calculé à partir d'une telle CAD sera constituée des points $x_1 < \cdots < x_p$ et d'un point par intervalle ouvert défini par cette subdivision de la droite réelle, ainsi que des signes que les polynômes de la famille $\mathcal{P}$ prennent en chacun de ces points.

Dans le cas univarié, on peut voir cet arbre (ici réduit à une liste) comme un *tableau de signes* w pour la famille $\mathcal{P}$. Un tel tableau est donné par une subdivision $x_1 < \cdots < x_p$ de la droite réelle $]-\infty, +\infty[$ telle que pour tout intervalle $I \in \{]-\infty; x_1[, [x_1],]x_1; x_2[, \ldots,]x_p; +\infty[\}$, pour tout $j \in \{1, \ldots, s\}$. Pour tout $x \in I$, $P_j(x)$ a pour signe $w(I, j)$, le signe se trouvant dans la case du tableau w dont les coordonnées sont I et j .

On cherche ainsi à obtenir une subdivision de $\mathbb{R}$, déterminée par l'ensemble $x_1, \ldots, x_n$ de toutes les racines des polynômes de la famille et les $2n + 1$ colonnes du tableau de signes seront indexées alternativement par les racines des polynômes, et par les intervalles qui les séparent. Si le tableau de signes n'a qu'une seule colonne, tous les polynômes sont des polynômes constants.

Par exemple, pour la famille $\{(X-1)(X-2), (X-2)(X-3)\}$, un tel tableau de

signes peut être :

	$]-\infty;1[$	1	$]1;2[$	2	$]2;3[$	3	$]3,+\infty[$
$(X-1)(X-2)$	+	0	−	0	+	+	+
$(X-2)(X-3)$	+	+	+	0	−	0	+

La représentation arborescente de ce tableau est donnée figure 3.2.

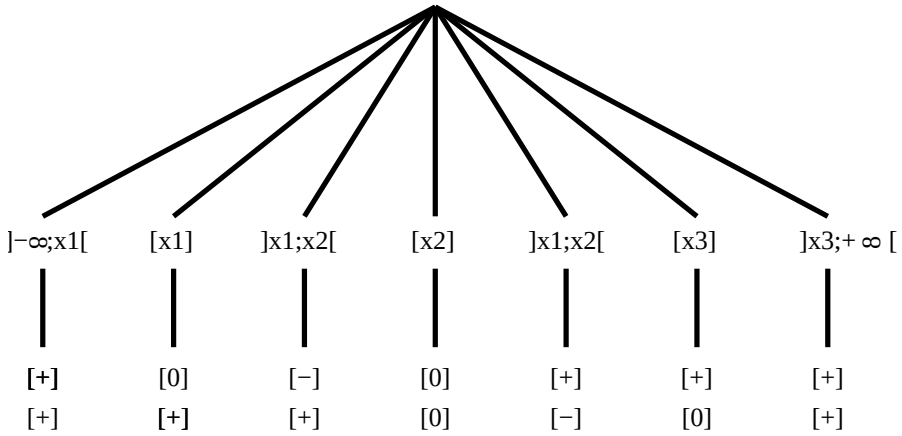


FIG. 3.2: *Arbre des signes de* $\{(X-1)(X-2),(X-2)(X-3)\}$

Dans cet arbre l'étiquette x_1 représente le point 1, x_2 le point 2 et x_3 représente le point 3. Néanmoins, dans le cas général, ces points seront algébriques et nous y ferons donc référence par des noms de variables.

Si l'on sait calculer un tableau de signes w pour $\mathcal{P}$, il est facile de décider une $\mathcal{P}$-formule. La formule $\forall x, B(x)$ est vraie si toutes les colonnes de w satisfont la condition $B(x)$. La formule $\exists x, B(x)$ est vraie si au moins une colonne de w satisfait la condition $B(x)$.

Le problème général

Dans le cas général des formules faisant intervenir des polynômes à plusieurs variables, on voit apparaître la structure arborescente et son utilité dans le problème de décision. La hauteur de ces arbres est égale au nombre de variable du problème considéré. A chaque étage k de l'arbre, le nombre de noeuds correspond à une analyse par cas sur la position de la $k$ème coordonnée x_k.

Considérons les deux familles réduites à un seul polynôme $\{XY\} \subset \mathbb{Q}[X,Y]$ et $\{X+Y\} \subset \mathbb{Q}[X,Y]$.

Les arbres de la figure 3.3 sont respectivement des arbres de conditions de signes cylindriques pour ces deux polynômes.

Revenons au cas de la formule $\exists x \forall y P(x,y) = 0$. Les différences de forme des arbres ci-dessus illustrent les différences qui conduisent la formule à être vraie ou non selon le polynôme choisi. En suivant un chemin dans les arbres, à partir de la racine :

– A gauche, quel que soit le noeud choisi à la profondeur 1, on pourra aboutir à une feuille de signe +.

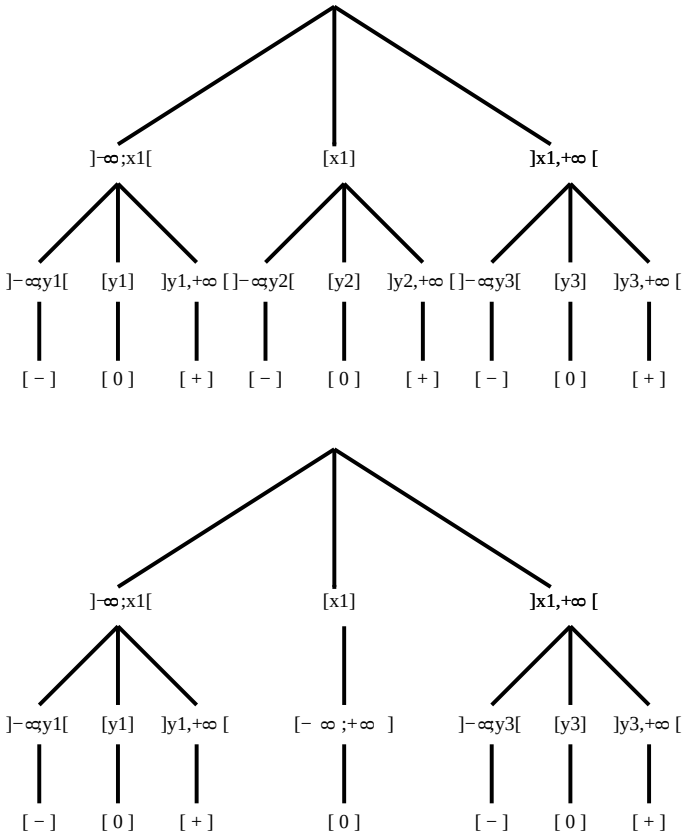


FIG. 3.3: *Arbres des signes de* $\{X + Y\}$ *et* $\{XY\}$

– A droite, en choisissant le noeud étiqueté par $[x_1]$, on ne peut aboutir qu'à une feuille portant l'étiquette [0].

Finalement, la formule $\exists x \forall y P(x, y) = 0$ est vraie si $P = XY$, car il existe dans tout échantillon cylindrique issu d'une CAD de XY de un noeud de hauteur 1 pour lequel toutes les feuilles sous ce noeud sont étiquetées par 0. Par contre la formule est fausse pour $P = X + Y$ car il n'existe pas de tel noeud pour un échantillon de points d'une CAD de $X + Y$.

Explosion combinatoire

Sur l'arbre représentant un échantillon cylindrique de points pour $X + Y$ représenté en figure 3.3, on peut déjà observer que l'on calcule bien plus de points (9) que de composante connexes (3). Par contre dès que l'on cherche un échantillon cylindrique, on calculera toujours ces (9) cellules, introduite par la projection.

Il est également important de choisir un ordre pertinent sur les variables (voir section 3.2.2).

3.1.4 Procédure de décision

Nous définissons inductivement la représentation CSarbre d'un arbre de conditions de signes comme :

– une Feuille étiquetée par une colonne de signes, ou bien
– un Noeud étiqueté par un intervalle réel I à bornes rationnelles, et une liste $t_1, \ldots, t_n$ de CSarbres.

Les colonnes de signes étiquetant les feuilles sont des n-uplets ordonnés de couples polynôme et signe, attribuant un signe à chacun des éléments de $\mathcal{P}$. Dans les figures, on omettra toujours les polynômes dans les étiquettes des feuilles, tant qu'il n'y a pas d'ambiguïté. Si A est un $\mathcal{P}$-atome de la forme $= P > 0$ (resp. $P = 0$, $P < 0$), on dira que la colonne de signes c valide A si et seulement si dans c, on trouve le couple $(P, +)$ (resp $(P, 0)$, $(P, -)$).

Étant donné un arbre t de conditions de signes cylindriques de la famille $\mathcal{P}$, on peut décider une $\mathcal{P}$-formule arbitraire F en suivant l'algorithme décrit en figure 3.4

L'algorithme de décision effectue en fait un parcours de l'arbre, à la recherche du motif décrit par la formule, qui est un chemin (si la formule est un atome précédé de quantifications existentielles) ou un ensemble de chemins.

Un algorithme de décision pour la théorie des réels va consister à calculer un arbre de conditions de signes cylindriques étant donnée une famille finie initialement fixée de polynômes.

3.1.5 Méthode de Hörmander

L'algorithme (Hörmander 1983) qui fait l'objet de cette section est certainement le plus élémentaire connu pour résoudre ce problème de décision. L'auteur s'est inspiré d'un manuscrit non publié de Paul Cohen datant de 1967. Il s'agit d'une procédure remarquable par sa simplicité, qui ne fait intervenir que des divisions euclidiennes et des variantes du

Cad_decision F t :=
Par cas sur F qui peut être :
- un atome $P\sharp 0$, (avec $\sharp \in \{>, <, =\}$) :
  si t = Feuille c alors
    si c valide $P\sharp 0$ alors **répondre** vrai
    sinon **répondre** faux
  sinon **échouer**
- $F_1 \wedge F_2$: **répondre** (Cad_decision F_2 t) et (Cad_decision F_1 t)
- $F_1 \vee F_2$: **répondre** (Cad_decision F_2 t) ou (Cad_decision F_1 t)
- $\neg F$: **répondre** *non* (Cad_decision F t)
- $\forall x,\ F$:
  si t = Noeud I t_1 ... t_n alors
  répondre (Cad_decision F t_1) et ... et (Cad_decision F t_n)
  sinon **échouer**
- $\exists x,\ F$:
  si t = Noeud I t_1 ... t_n alors
  répondre (Cad_decision F t_1) ou ... ou (Cad_decision F t_n)
  sinon **échouer**

FIG. 3.4: *Procédure de décision à partir d'un arbre de conditions de signes*

théorème des valeurs intermédiaires. On en trouve également un exposé détaillé dans l'ouvrage de Bochnak et al. (Bochnak, Coste, et Roy 1998).

Le problème à une variable

On se trouve donc dans le cas où on cherche le *tableau de signes* w d'une famille de polynômes $\mathcal{P}$. En effet, un arbre de profondeur 1 peut être représenté par un tableau dont les lignes seraient indexées par les polynômes de la famille et les colonnes, par les intervalles de la partition. Dans ce tableau, les colonnes contiennent les signes d'un polynôme sur un intervalle.

La méthode repose sur les remarques suivantes. Le tableau des signes d'une famille de polynômes tous constants est facile à construire. Puis, si $\mathcal{P} = \{f_1, \ldots, f_s\}$ est la famille de polynômes dont on veut construire le tableau, on peut reconstruire le tableau de signes de $\mathcal{P}$ à partir de celui de la famille où f_s a été supprimé et remplacé par sa dérivée et les restes des pseudo-divisions euclidiennes respectives de f_s par les f_i et f'_s. La pseudo-division est nécessaire puisqu'on veut conserver des polynômes à coefficients dans l'anneau des entiers.

Cette transformation sur $\mathcal{P}$ fait grossir le nombre de polynômes de la famille, mais si on choisit de supprimer toujours le polynôme de plus haut degré, soit le degré maximal des éléments diminue, soit le nombre de polynômes qui ont ce degré diminue. L'application successive de cette transformation aboutira ainsi en temps fini à une famille de polynômes tous constants.

L'algorithme se déroule ainsi : soit $\mathcal{P} = \{f_1, \ldots, f_s\} \subset \mathbb{Q}[X]$. On suppose que les f_i sont non identiquement nuls et que f_s est non constant, de degré maximal. Si l'on ne peut obtenir cette configuration par permutation dans la famille $\mathcal{P}$, alors la famille ne contient

que des polynômes constant et le calcul de son tableau est trivial.

Sinon, soit $a_1, \ldots, a_{s-1}$ les coefficients dominants respectifs de $f_1, \ldots, f_{s-1}$ et a_s le coefficient dominant de f'_s. Les relations de pseudo-divisions donnent :

$$\forall i = 1, \ldots s-1, \quad c_i f_s = q_i f_i + g_i, \quad deg(g_i) < deg(f_i)$$

où c_i divise une puissance de a_i. On effectue aussi la division de f_s par sa dérivée :

$c_s f_s = q_s f'_s + g_s, \quad deg(g_s) < deg(f'_s)$ où c_s divise une puissance de a_s.

Soit w' le tableau des signes de $\{f_1, \ldots, f_{s-1}, f'_s, g_1, \ldots g_s\}$ et $x_1 < \cdots < x_N$ la subdivision associée.

Soit $z_1 < \cdots < z_p$ les x_i qui sont racines de $f_1, \ldots, f_{s-1}$ ou f'_s. Des équations de pseudo-divisions on déduit le fait qu'en un z_i racine de f_j (resp. f'_s), f_s a le signe de g_j (resp. g_s). Entre les z_i, le théorème des valeurs intermédiaires donne éventuellement des racines supplémentaires pour f_s, mais une au plus par intervalle car f'_s y est non nulle et de signe constant.

Par exemple, supposons que :

- z_1 est racine de f_j : c'est à dire que $w([z_1], f_j) = 0$
- z_2 est racine de f_k : c'est à dire que $w([z_2], f_k) = 0$
- f'_s est positive sur $]-\infty; x_1[$: c'est à dire que $w'(]-\infty, x_1[, s) = +$.

Alors :

- Si $w'([z_1], g_j) = +$ et $w'([z_2], g_k) = -$, alors il existe un unique $y \in]z_1, z_2[$ tel que $f_s(y) = 0$. En effet, f'_s ne s'annule pas sur $]z_1, z_2[$, donc comme elle est continue, elle garde un signe constant. De plus, $g_j(z_1) > 0$ et $g_k(z_2) > 0$ et on en déduit que $f_s(z_1) > 0$ et $f_s(z_2) < 0$. Comme f_s est une fonction continue strictement monotone sur $]z_1, z_2[$, le théorème des valeurs intermédiaires assure le résultat.
- Si $w'([z_1], g_j) = +$ et $w'([z_2], g_k) \neq -$, alors on peut déduire que f_s ne s'annule pas sur $]z_1; z_2[$ car elle est monotone et continue.
- Si $w'([z_1], g_j) \neq +$, alors on peut en déduire que f_s ne s'annule pas sur $]-\infty, z_1[$

Avec des raisonnement analogues, on peut couvrir tous les cas et les positions relatives des racines de f_s et son signe.

Soit $y_1 < \cdots < y_m$ la réunion des racines de f_s données par le théorème des valeurs intermédiaires et des z_i non racines de f'_s. Ces points donnent la subdivision du tableau w recherché. En effet on a obtenu la liste exhaustive des racines de la famille $\mathcal{P}$ puisqu'on avait déjà celles de $\{f_1, \ldots, f_{s-1}\}$ et qu'on a trouvé et intercalé toutes celles de f_s. Les signes de $f_1, \ldots, f_{s-1}$ sont tirés directement du tableau w'. Ceux de f_s sont déduits de signes de f'_s et des signes des g_i dans w'.

Le problème général à plusieurs variables

Dans la cas du problème général, la méthode reste identique. Il s'agit d'abord de fixer un ordre sur les indéterminées. Les polynômes à n indéterminées, éléments de $\mathbb{Q}[X_1, \ldots, X_n]$ seront considérés comme des polynômes à une indéterminée, à coefficients dans un anneau de polynômes, c'est à dire comme des éléments de $\mathbb{Q}[X_1, \ldots, X_{n-1}][X_n]$.

Néanmoins, il faut tenir compte à présent du problème de nullité éventuelle des coefficients dominants des polynômes manipulés lors des transformations successives. Ces coefficients sont à présent des polynômes en $X_1, \ldots, X_{n-1}$ qui peuvent s'annuler selon les valeurs qui instancient ces $n-1$ premières variables.

On effectue ainsi les calculs en gardant en mémoire une liste $\mathcal{N}$ de polynômes qu'on suppose non nuls : ce sont en fait les coefficients dominants non constants qui interviennent dans les pseudo-divisions. Au départ $\mathcal{N}$ est vide.

Après un certain nombre d'étapes de transformation décrites dans le cas à une variable, tous les polynômes de la famille seront *constants en* X_n. On va alors calculer les tableaux de signes possibles pour cette famille vue comme une famille de polynômes à une variables X_{n-1} (avec des coefficients dans $\mathbb{Q}[X_1, \ldots, X_{n-2}]$, éventuellement réduit à $\mathbb{Q}$). Dans ces tableaux, on va sélectionner les colonnes de signes où les polynômes de la famille qui divisent des polynômes se trouvent dans $\mathcal{N}$ ont effectivement un signe différent de 0.

Lorsqu'un des polynômes de $\mathcal{P}$ est non constant, on suit la procédure suivante :

- soit c le premier coefficient dominant non constant de $f_1, \ldots, f_s$ qui ne divise pas un des polynômes de $\mathcal{N}$. C'est à dire que ce coefficient peut être nul alors que les polynômes dans $\mathcal{N}$ sont non nuls. Soit $f_i = cX^d + r$ le polynôme dont c est le coefficient dominant. On traite alors les deux cas :
  - $\{f_1, \ldots f_{i-1}, f_i, f_{i+1}, \ldots f_s\}$ en ajoutant les facteurs sans carrés de c à $\mathcal{N}$.
  - $\{f_1, \ldots f_{i-1}, r, f_{i+1}, \ldots f_s\}$ dont on ne garde que les tableaux de signes où c est identiquement nul.
- soit il n'y a pas de tel c. Dans ce cas les polynômes de $\mathcal{N}$ sont non nuls, et on peut appliquer la méthode présentée sur le cas à une variables (dérivation et pseudo-divisions) car les conditions de bonne formation des polynômes sont respectées.

On calcule donc en fait les arbres de conditions de signe pour des familles beaucoup plus grandes que celle qui nous intéresse initialement.

La complexité de cet algorithme est considérable : c'est une tour d'exponentielles de hauteur le nombre de variables du problème. Néanmoins, un travail peu coûteux d'optimisation peut améliorer sensiblement les performances de l'algorithme dans la plupart des cas.

Par exemple, il est très utile de factoriser les polynômes de la famille initiale $\mathcal{P}$ pour extraire leur facteurs sans carrés. Cette opération est peu coûteuse puisqu'elle ne demande que des calculs de pgcd (effectués grâce à l'algorithme des sous-résultants puisque les coefficients des polynômes sont eux-mêmes des polynômes). Il est également très facile de recomposer les signes des éléments de $\mathcal{P}$ à partir de ceux de leur facteurs.

Un exemple de calculs

Supposons qu'on s'intéresse à la famille $\mathcal{P} = \{XY^3 + Y^2\} \subset \mathbb{R}[X][Y]$ réduite à un unique polynôme. Initialement, la liste $\mathcal{N}$ est vide.

La factorisation sans carrés de f donne deux facteurs, qui constituent une nouvelle famille $\mathcal{P} = \{XY + 1, Y\}$. Il faut en étudier les coefficients dominants non constants.

- $X \neq 0$: maintenant $\mathcal{N} = \{X\}$, et tous les coefficients dominants de f sont non nuls. On peut appliquer la méthode à une variable :
  - $f_1 = Y, f_2 = XY + 1$ puis $f_2' = X, g_1 = 0, g_2 = 1$. La factorisation sans carrés est programmée pour éliminer les polynômes constants : la famille courante f devient $\{Y, X\}$.
  - Les coefficients dominants sont tous non nuls, on peut dériver et on obtient la famille $\{X, 1, 0, 0\}$. En factorisant, on est ramené à la famille $\{X\}$, constante en Y.

– On traite $\{X\}$ comme une famille de polynômes en X. En dérivant, on obtient la famille $\{1, 0\}$, dont tous les polynômes sont constants et dont le tableau de signes est :

1	+
0	0

On en déduit celui de $\{X\}$:

X	−	0	+

Il y a donc trois tableaux de signes possibles pour $\{X\}$ considéré maintenant comme une famille de polynômes (constants) en Y.

X	−

X	0

X	+

On n'en retient que les cas où X est non nul, puisque X divise un de polynômes de $\mathcal{N} = \{X\}$:

X	−

X	+

En remontant à la famille $\{Y, X\}$, on a donc deux tableaux :

Y	−	0	+
X	−	−	−

Y	−	0	+
X	+	+	+

Puis on remonte à la famille $\{XY+1, Y\}$, qui a également deux tableaux de signes :

$XY+1$	+	+	+	0	−
Y	−	0	+	+	+

$XY+1$	−	0	+	+	+
Y	−	−	−	0	+

ce qui achève le cas $X \neq 0$

– $X = 0 : \mathcal{N} = \{\}$ et la famille courante devient $\{1, Y\}$ puis $\{Y\}$ après factorisation des sans carrés. On obtient un unique tableau de signes : $\{Y : [-, 0, +]\}$ qui est valide puisque ses polynômes ne divisent aucun des polynômes de $\mathcal{N}$. D'où un tableau de signes pour $\{XY + 1, Y\}$ quand X est nul :

$XY+1$	+	+	+
Y	−	0	+

Récapitulons les trois tableaux de signes possibles pour la famille $\{XY + 1, Y\}$:

$XY+1$	+	+	+	0	−
Y	−	0	+	+	+

$XY+1$	−	0	+	+	+
Y	−	−	−	0	+

$XY+1$	+	+	+
Y	−	0	+

D'où les trois tableaux possibles pour la famille de départ $\{XY^3 + Y^2\}$:

XY^3+Y^2	+	0	+	0	−

XY^3+Y^2	−	0	+	0	+

XY^3+Y^2	+	0	+

Finalement, l'arbre de conditions de signes cylindriques est construit en "oubliant" les étapes intermédiaires de transformation qui n'éliminent pas de variables, et en ne gardant que les étapes de passages d'une variable à une autre. Ces étapes sont celles qui introduisent l'arborescence.

Sur cet exemple, l'arbre est celui représenté par la figure 3.5

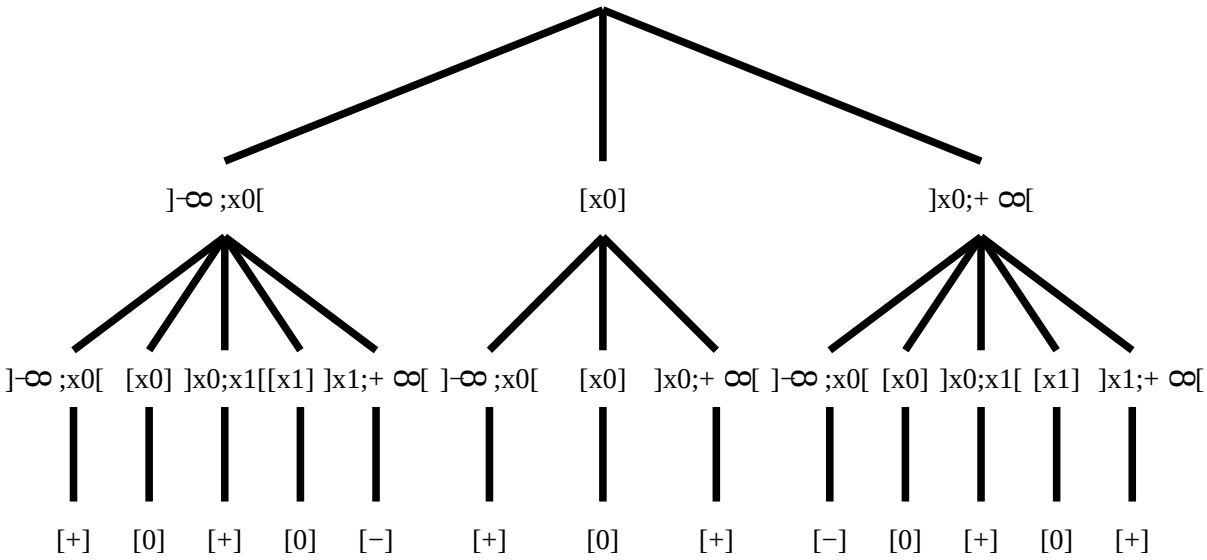


FIG. 3.5: *Arbres des signes de* $XY^3 + Y^2$

Les points des subdivisions étiquetant les noeuds internes sont les racines de familles de polynômes à une variable.

Deux expériences dans les systèmes de preuves

La méthode de Hörmander a été utilisée à deux reprises pour la construction d'un outil d'automatisation dans un système de preuves.

Une implémentation de cet algorithme en Ocaml a servi de support pour programmer un premier prototype de tactique pour les réels. Cette implémentation fournissait à Coq une trace des calculs effectués (divisions euclidiennes, dérivations, factorisations) pour que la tactique recompose une preuve de la réponse de la procédure de décision pour chaque instance. Dans ce travail (Mahboubi et Pottier 2002), il n'était donc pas question de donner une preuve formelle de la correction de l'algorithme.

Malheureusement, la recomposition de la preuve est très coûteuse dans cette approche. En particulier la lenteur de la version alors disponible de la tactique `ring`, utilisée intensivement pour vérifier les factorisations et les identités de division euclidienne était un facteur extrêmement limitant. D'autre part la complexité intrinsèque de l'algorithme est énorme (une tour d'exponentielle de hauteur le nombre de variables des polynômes) et limite très rapidement les problèmes que l'on peut traiter avec cette tactique à des énoncés très simples.

À la suite de ces travaux, J. Harrison et S. McLaughlin ont également utilisé la méthode de Hörmander pour doter HOL-Light d'une tactique pour les nombres réels

(McLaughlin et Harrison 2005).

3.2 Décomposition Algébrique Cylindrique

Dans cette section, nous présentons l'algorithme de décomposition algébrique cylindrique (CAD). Cet algorithme est issu des travaux de G.E. Collins (Collins 1975). La parution de ces travaux constitua un tournant majeur dans le domaine de la géométrie algébrique réelle. En effet les algorithmes d'A. Tarski ainsi que les variantes postérieures parues jusque là avaient une complexité non exprimable par une fonction récursive (des tours d'exponentielles de hauteur variable). L'algorithme de Collins quant à lui est "seulement" doublement exponentiel en le nombre de variables et polynomial en le degré.

Une amélioration aussi drastique dans la complexité de cet algorithme de décision ouvrait de nouvelles perspectives de recherche en géométrie algébrique réelle, dans lesquelles l'ordinateur constituerait une aide pour s'attaquer à des problèmes difficiles. Par exemple, la première partie du 16ème problème de Hilbert, qui propose de classifier topologiquement les courbes algébriques réelles planes reste un problème ouvert et un domaine actif de recherche.

Malheureusement, la complexité de l'algorithme de Collins ne permet de traiter qu'une classe très restreinte de problèmes, de bas degré et de petite dimension, ce qui rend la théorie des corps réels clos discrets décidable mais en pratique non calculable sur des problèmes de grande envergure. Il n'existe pas de borne connue sur la complexité intrinsèque du problème de décision, mais les algorithmes de décision complète que l'on sait implémenter ont la même classe de complexité en pire des cas que l'algorithme de Collins.

Outre les implémentations d'algorithmes de décomposition algébrique cylindrique à proprement parler, des outils de calcul formel sophistiqués ont néanmoins été développés autour de techniques symboliques/numériques issues de la CAD (Aubry, Rouillier, et Safey 2002), qui ont des applications par exemple en robotique (Corvez 2005), ou en théorie du contrôle (Ying, Xu, et Lin 1999; Hong, Liska, et Steinberg 1997)

Rechercher de nouveaux algorithmes de complexité praticable permettant une étude effective des ensembles semi-algébrique reste un défi stimulant (Basu, Pollack, et Roy 2006) car à l'heure actuelle les méthodes connues (comme la CAD ou les algorithmes de déformation des variétés) donnent des résultats équivalents en terme d'efficacité.

Les motivations pour implémenter un tel algorithme dans un assistant à la preuve sont assez différentes de celles d'un utilisateur de système de calcul formel. En effet dans un système de calcul formel, on attend de la machine qu'elle fournisse des résultats de calculs trop long et trop gros pour être effectués par un être humain en temps raisonnable. Dans un assistant à la preuve, on veut disposer d'un outil d'automatisation qui soulage l'utilisateur des preuves formelles longues et pénibles de résultats que l'on prouve rapidement sur une feuille de papier.

C'est pourquoi les exigences en complexité sont différentes : on verra que la complexité de l'algorithme de Collins nous permet déjà de prouver formellement en quelques secondes des énoncés dont les preuves demanderaient des heures de travail sans cet outil. Dans notre contexte, le temps de calcul n'est pas le seul à prendre en compte : ce qui nous intéresse est le temps de construction de la preuve (comme terme Coq). C'est l'intérêt

de la méthode réflexive calculatoire que nous allons employer ici (voir chapitre 4), selon laquelle le temps de calcul *est* le temps de preuve.

Il existe dans la littérature de nombreuses descriptions de l'algorithme de Collins. C'est pourquoi dans l'exposé ci-dessous nous essayons d'adopter un point de vue de programmeur, qui est lui plus rare dans les publications au sujet de la CAD, en présentant l' algorithme de façon granulaire : on commencera par expliquer le principe général de la méthode, en donnant progressivement les détails des algorithmes qui permettent de réaliser les différents ingrédients. La dernière section du chapitre est consacrée à des exemples concrets de calculs.

Nous omettons la plupart du temps les preuves des propriétés énoncées dans la suite de ce chapitre, dans un souci de concision et de clarté. Les preuves omises se trouvent sauf mention du contraire dans l'ouvrage (Basu, Pollack, et Roy 2006).

3.2.1 Présentation schématique de la méthode

Soit $\mathcal{P} \subset \mathbb{Q}[X_1, \ldots, X_n]$ une famille de polynômes. L'algorithme de décomposition algébrique cylindrique calcule un échantillon (cylindrique) de points de l'espace qui représente une partition de $\mathbb{R}^n$ sur laquelle chacun des polynômes de la famille de départ a un signe invariant.

Tout d'abord, on dispose d'un algorithme d'*isolation des racines réelle*, qui opère sur une famille de polynômes de $D[X]$ où D est un sous-anneau de $\mathbb{R}$.

Définition 3.2.1 (Liste d'isolation) *Soit D un sous-anneau de $\mathbb{R}$, et une liste de polynômes $P_1, \ldots, P_s \in D[X]$. Soit $Roots_{\mathcal{P}}$ l'ensemble des racines réelles de cette famille de polynômes :*

$$Rac_{\mathcal{P}} := \{x \in \mathbb{R} \quad | \quad \exists i,\ P_i(x) = 0\}$$

Une liste d'isolation *pour $\mathcal{P}$ est une liste L de sous-ensembles de $\mathbb{R}$ composée de singletons rationnels et d'intervalles ouverts à bornes rationnelles. Chaque élément de L contient exactement un élément de $Rac_{\mathcal{P}}$ et il y a autant d'éléments dans L que dans $Rac_{\mathcal{P}}$.*

Puis on construit un échantillon représentatif des signes de la famille sur la droite réelle en ajoutant à la liste d'isolation un échantillon de points intermédiaires, les milieux des bornes de deux éléments consécutifs de la liste d'isolation ainsi qu'un majorant strict et un minorant strict des points de l'échantillon.

Enfin il faut calculer le signe de tous les polynômes en chacun des points de cet échantillon exhaustif.

La remarque suivante est importante : c'est le cahier des charge de la procédure de construction de la liste d'isolation d'une famille de polynômes à une variable, qui sera le support de l'algorithme général.

Remarque 3.2.1.1 (Prérequis pour l'isolation des racines réelles) *Soit D un sous-anneau de $\mathbb{R}$, et une liste de polynômes $\{P_1, \ldots, P_s\} \subset D[X]$.*

Supposons que :

– on sait évaluer le signe d'un coefficient, c'est à dire d'un élément $d \in D$;

– *on sait évaluer le signe d'un polynôme $T \in D[X]$ quelconque en un point rationnel.*

Si ces conditions sont remplies, alors

– *lorsque le polynôme $T \in D[X]$ a une unique racine réelle dans l'intervalle $]a, b[$ à bornes rationnelles, on saura évaluer le signe d'un polynôme S arbitraire en cette racine de T ;*
– *on saura calculer une liste d'isolation pour $\mathcal{P}$;*
– *on saura calculer un échantillon exhaustif comme décrit ci-dessus ainsi que toutes les conditions de signes réalisées en les éléments de cet échantillon.*

Lorsque le problème général est un problème à une variable (cas $n = 1$), ces conditions nécessaires seront remplies. On verra en section 3.2.4 comment réaliser ces algorithmes.

Le traitement des problèmes en dimension supérieure se fait par élimination successive des variables, ce qui correspond à des projections successives de la variété.

Soit $\mathcal{P} \subset \mathbb{Q}[X_1, \ldots, X_n]$.

$$\begin{array}{ccc} \mathbb{R}[X_1, \ldots, X_n] & & \mathbb{R}[X_1, \ldots, X_{n-1}] \\ \\ \mathcal{P} = P_1, \ldots, P_s & \xrightarrow[\text{d'élimination}]{\text{phase}} & \mathcal{Q} = Q_1, \ldots, Q_t \\ \\ \text{CAD et signes pour } \mathcal{P} & \xleftarrow[\text{remontée}]{\text{phase de}} & \text{CAD et signes pour} \mathcal{Q} \\ \\ \mathbb{R}^n & & \mathbb{R}^{n-1} \end{array}$$

$\mathcal{Q}$ est une nouvelle famille de polynômes, qui peut être plus nombreuse que $\mathcal{P}$, mais dans laquelle X_n a été éliminé. On distingue deux phases dans ce procédé récursif.

Durant la *phase d'élimination*, partant d'une famille $\mathcal{P} \subset \mathbb{Q}[X_1, \ldots, X_n]$, on calcule une famille $\mathcal{Q} \subset \mathbb{Q}[X_1, \ldots, X_{n-1}]$, en appliquant à $\mathcal{P}$ un *opérateur de projection*, qui va éliminer la variable X_n. Cette phase calcule successivement les familles $proj_i(\mathcal{P}) \subset \mathbb{Q}[X_1, \ldots, X_i]$ pour $i = n \ldots 1$, en partant de $proj_n(\mathcal{P}) = \mathcal{P}$.

La propriété de correction de cet opérateur de projection s'exprime de la façon suivante : étant donnée une partition $\mathcal{S}_{n-1}$ adaptée à la nouvelle famille $\mathcal{Q}$, on sait calculer une partition $\mathcal{S}_n$ adaptée à la famille de départ $\mathcal{P}$. Une cellule $C_\mathcal{P}$ de cette partition adaptée à $\mathcal{P}$ sera obtenue en :

– choisissant une cellule $D_\mathcal{Q}$ issue de la partition adaptée à $\mathcal{Q}$;
– considérant le cylindre "à bords polynomiaux" dans ce cylindre :
  – soit en prenant l'intersection de ce cylindre avec le graphe d'une fonction semi-algébrique ;
  – soit en considérant le demi cylindre au-dessous (resp. au dessus) d'une fonction semi-algébrique ;
  – soit en considérant la portion de cylindre comprise entre les graphes respectifs de deux fonctions semi-algébriques.

Les éléments de $\mathcal{S}_n$ sont appelées cellules de niveau n et l'ensemble des partitions successives $\mathcal{S}_1 \ldots \mathcal{S}_n$ est appelé *décomposition algébrique cylindrique* de $\mathbb{R}^n$.

Voici par exemple représentée sur la figure 3.6 une décomposition algébrique cylindrique adaptée à la sphère en dimension 3.

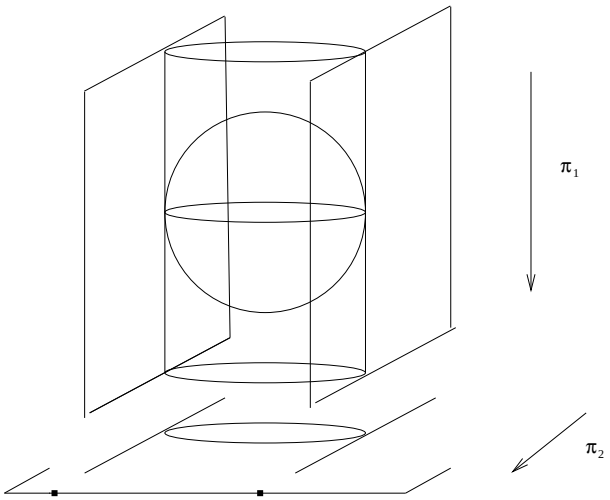


FIG. 3.6: *CAD adaptée à la sphère en dimension* 3

Les cellules de niveau un sont constituées de deux points, de deux demi-droites et d'un intervalle ouvert. Dans les cellules de niveau deux, on trouve le disque ouvert, projeté de l'intérieur de la sphère. Les cellules de niveau trois issues du cylindre qui a ce disque pour section sont les deux hémisphères qui forment l'intersection du cylindre et de la sphère, l'intérieur de la sphère, et les deux demi-cylindres infinis s'appuyant sur la sphère.

Le cellules de niveau $n-1$ étant données, le calcul des cellules de niveau $n-1$ se fait au cours de la *phase de remontée*. Le fait que l'on puisse obtenir ces cellules de niveau n comme des portions de cylindres délimitées par des *fonctions* polynomiales provient du fait que le comportement des polynômes de $proj_{i+1}(\mathcal{P})$ est "uniforme" sur les cellules de niveau i, par rapport aux i premières coordonnées.

En fait, les portions de cylindres formant les cellules de niveau $i+1$ seront délimitées par des *fonctions* semi-algébriques implicitement définies par :

$$P(x_1, \ldots, x_i, f(x_1, \ldots, x_i)) = 0$$

où P est un polynômes de $proj_{i+1}(\mathcal{P})$ et $(x_1, \ldots, x_i) \in \mathbb{R}^i$ un point de la cellule de niveau i.

Donnons quelques exemples des situations possibles. Une droite $\overline{x} \times \mathbb{R}$ devra couper la variété algébrique définie par les polynômes de $\mathcal{P}$ en le même nombre de points quel que soit le point $\overline{x}$ choisi dans une cellule de niveau $n-1$. La figure 3.7 illustre l'importance des points singuliers dans le calcul de la décomposition. C'est pour cette raison que l'on retrouve deux points dans les cellules de niveau 1 de la sphère (voir encore la figure 3.6).

Il faut aussi détecter les points d'annulation des coefficients dominants des polynômes. En effet cette annulation induit une modification du degré du polynôme et un changement de coefficient dominant.

Par exemple, si dans la famille de départ, on trouve le polynôme $P(X, Y) = YX^2 + X$ et que l'ordre fixé sur les variables élimine d'abord la variable Y, il faudra faire en sorte

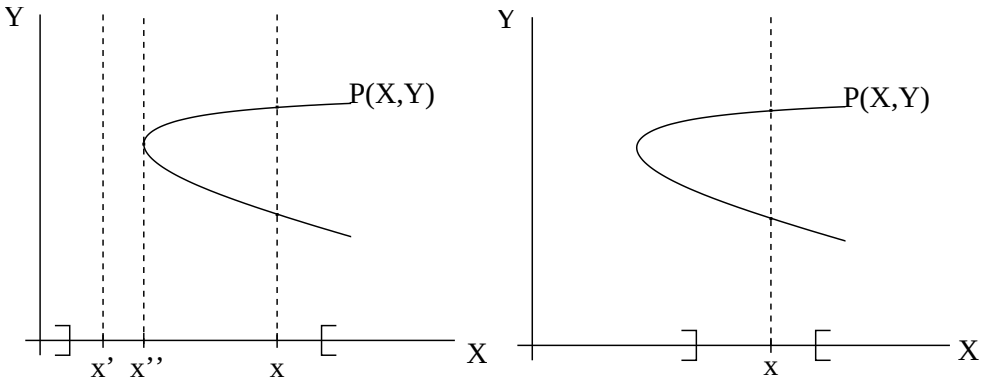


FIG. 3.7: *À gauche, l'intervalle ne peut pas être une cellule de niveau* 1 *:* $P(x, Y)$ *a deux racines,* $P(x', Y)$ *n'a pas de racines et* $P(x'', Y)$ *a une unique racine. À droite, l'intervalle est un bon candidat*

que chaque cellule de niveau 1 ne contienne pas simultanément les points 0 et 1. En effet si $y = 0$, $P(X, y) = X$, un polynôme de degré 1 et de coefficient dominant 1, et si $y \neq 0$, $P(X, y) = yX^2 + X$, un polynôme de degré 2 et de coefficient dominant y. On observe ce phénomène sur la figure 3.8 qui représente la courbe algébrique définie par $YX^2 + X = 0$.

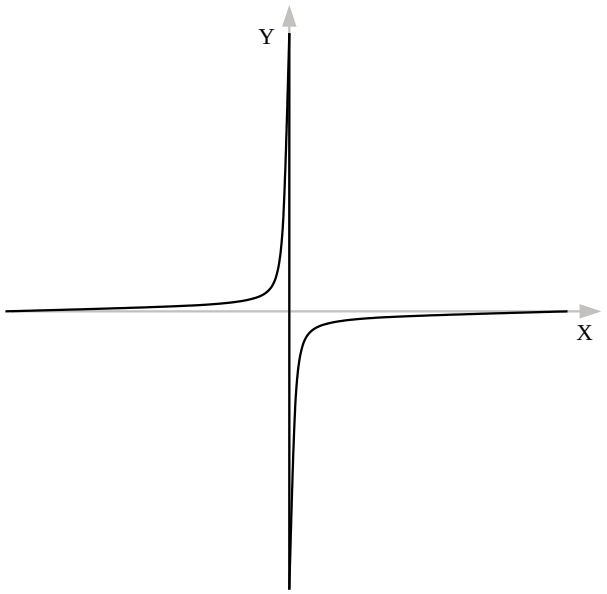


FIG. 3.8: *La courbe* $YX^2 + X = 0$ *: il y a un unique point sur la courbe d'ordonnée* $y = 0$*. Pour* $y \neq 0$*, il y a deux points sur la courbe d'ordonnée* y

Une autre configuration qui détermine les frontières entre cellules est l'intersection de plusieurs courbes, comme illustré sur la figure 3.9.

On pourra ainsi être capable d'inférer le comportement de n'importe quel polynôme P de $\mathcal{P}$ sur toute la cellule $D_{\mathcal{Q}}$ en étudiant le comportement de $P(a_1, \ldots, a_{n-1}, X)$, où $(a_1, \ldots, a_{n-1})$ est un point arbitrairement choisi dans un cellule $D_{\mathcal{Q}}$. Ce point témoin $(a_1, \ldots, a_{n-1})$ aura été récursivement calculé par l'algorithme.

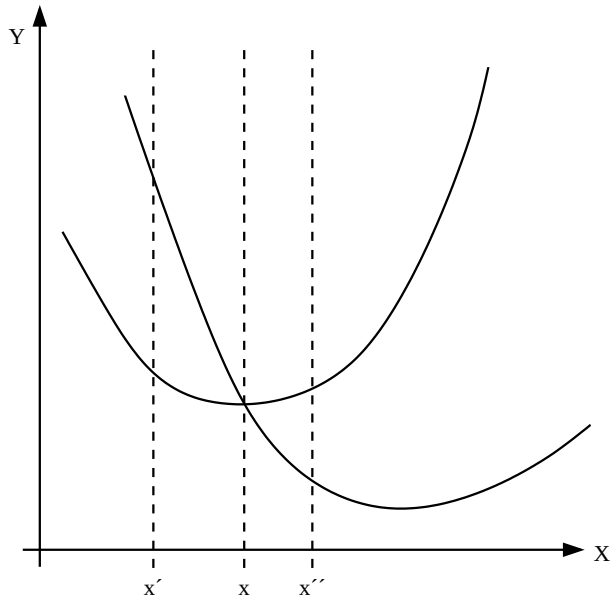


FIG. 3.9: *Les points x, x' et x'' ne doivent pas se trouver dans la même cellule de niveau 1*

3.2.2 Phase d'élimination

La proposition suivante, qui provient de la continuité des racines d'un polynôme, permet de préciser à quelles conditions les racines d'un famille de polynômes au dessus d'un ensemble semi-algébrique définissent une structure cylindrique.

Proposition 3.2.1 *Soit $\mathcal{P}$ un sous-ensemble fini de $\mathbb{Q}[X_1, \ldots, X_n]$ et soit S un sous-ensemble semi-algébrique connexe de $\mathbb{R}^{n-1}$. On suppose que pour tout $P \in \mathcal{P}$,*

- *$deg(P(\overline{x}, X))$ est indépendant du choix de $\overline{x}$ dans S ;*
- *le nombre de racines réelles distinctes de $P(\overline{x}, X)$ est indépendant du choix de $\overline{x}$ dans S ;*
- *pour tout $Q \in \mathcal{P}$, $deg(pgcd(P(\overline{x}, X), Q(\overline{x}, X)))$ est aussi indépendant du choix de $\overline{x}$ dans S.*

Alors, il existe l fonctions semi-algébriques continues $\xi_1 < \cdots < \xi_l : S \to \mathbb{R}$ telle que, pour tout $\overline{x} \in S$, l'ensemble des racines réelles de $\prod_{P \in \mathcal{P}'} P(\overline{x}, X)$, où $\mathcal{P}'$ est le sous-ensemble de $\mathcal{P}$ constitué des polynômes non identiquement nuls sur S de $\mathcal{P}$, est exactement $\{\xi_1(\overline{x}), \ldots, \xi_l(\overline{x})\}$. De plus, pour $i = 1 \ldots l$, et pour tout $P \in \mathcal{P}'$, la multiplicité de la racine $\xi_i(\overline{x})$ de $P(\overline{x}, X)$ est indépendante de $\overline{x}$.

Voir Proposition 5.14 pp 166 et suivantes dans (Basu, Pollack, et Roy 2006) □

Cette proposition énonce les conditions que doivent satisfaire les cellules de niveau $n-1$, conditions qui doivent être assurées par l'invariance des signes des polynômes de la famille $Elim_{X_n}(\mathcal{P})$.

Il existe plusieurs façons de concevoir un tel opérateur $Elim_{X_n}$, qui calcule une famille adéquate de polynômes dans $\mathbb{Q}[X_1, \ldots, X_{n-1}]$. L'opérateur que j'ai choisi de programmer est appelé *élimination par les sous-résultants*.

Il repose sur les propriétés des coefficients sous-résultants d'une paire de polynômes. Ces propriétés ainsi que le détail de la définition de cet opérateur sont donnés dans la section 3.2.5.

Ces fonctions semi-algébriques sont celles qui réalisent les fonctions ξ_j de la définition 3.1.7.

Remarque sur l'ordre d'élimination des variables

Selon l'ordre d'élimination choisi sur les variables présentes dans le problème initial, la complexité des calculs peut être bien différente car le nombre de cellules calculées peut changer dans une mesure importante.

Un exemple simple illustre ce phénomène. On s'intéresse à la CAD de la famille $\{(X+3)^2+(Y+1)^2-4, (X-3)^2+(Y-1)^2-4\}$. Les zéros de cette famille sont deux cercles de rayon 2, l'un centré en $(-3,1)$ et l'autre en $(3,1)$.

Les figures 3.10 et 3.11 représentent le décompte des cellules pour chacun des deux ordres de projection possibles.

Les points sur la droite au bas de la figure délimitent les cellules de niveau 1. Pour chacune de ces cellules de niveau 1, on matérialise le cylindre correspondant par une droite en pointillés, au dessus d'un point représentatif de la cellule de niveau 1.

Enfin, on indique le nombre de cellules de niveau 2 qui partitionnent le cylindre correspondant.

Sur la figure 3.10, on a d'abord projeté selon la direction Y, c'est à dire qu'on a éliminé la variable X en premier. Le nombre total de cellules de cette CAD est :

$$1+3+5+3+1+3+5+3+1=25$$

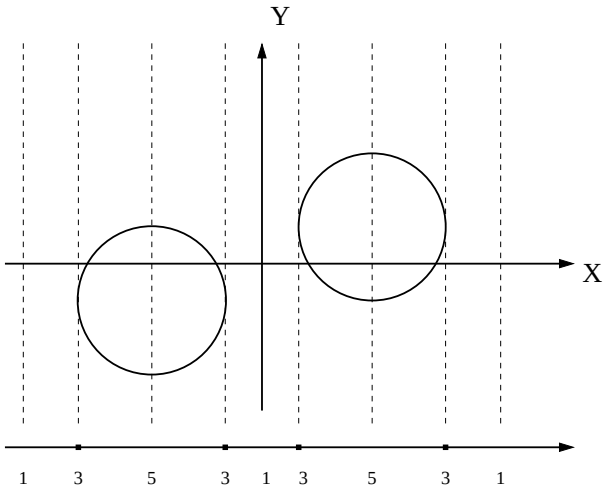


FIG. 3.10: *CAD de la famille* $\{(X+3)^2+(Y+1)^2-4, (X-3)^2+(Y-1)^2-4\}$. *On élimine* Y *puis* X

Sur la figure 3.11, on a d'abord projeté selon la direction X, c'est à dire qu'on a éliminé la variable Y en premier. Le nombre total de cellules de cette CAD est :

$$1+3+5+7+9+7+5+3+1=41$$

Il est bien évident qu'on préférera projeter dans l'ordre $Y > X$ plutôt que le contraire, car les calculs seront plus coûteux dans le second cas pour un résultat comportant autant d'information.

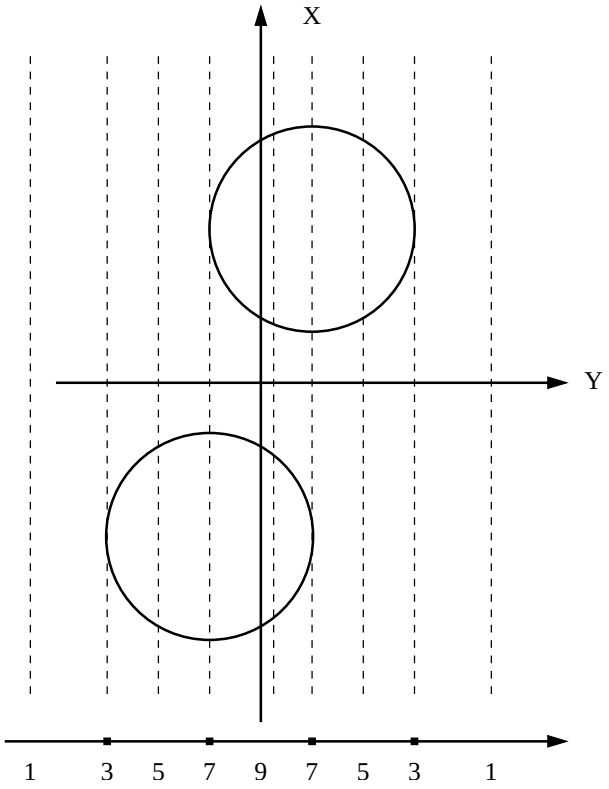


FIG. 3.11: *CAD de la famille* $\{(X+3)^2+(Y+1)^2-4, (X-3)^2+(Y-1)^2-4\}$. *On élimine* Y *puis* X

Le problème qui consiste à trouver un ordre pertinent sur les variables pour limiter le nombre de cellules calculées a été traité dans les travaux de A. Dolzmann, A. Seidl et T. Sturm (Dolzmann, Seidl, et Sturm 2004). Les auteurs ont dégagé à partir d'une étude statistique un moyen de choisir un ordre dans ces variables qui soit proche de l'ordre optimal, qui minimise le nombre de cellules.

Celui-ci repose en fait sur un procédé glouton, qui calcule à chaque étape d'élimination les familles issues de l'élimination de chacune des variables possibles, puis choisi la plus "petite".

Pour le moment je n'ai pas implémenté de calcul automatique et efficace d'un ordre de projection. Celui-ci est déterminé par l'occurrence des variables dans les polynômes de la famille de départ.

3.2.3 Phase de remontée

Lors de la phase de remontée, on calcule les cellules de niveau n à partir des cellules de niveau $n-1$. Plus précisément, à partir d'un échantillon de points de $\mathbb{R}^{n-1}$ comportant un point par cellule de niveau $n-1$, on va calculer un nouvel échantillon de points de $\mathbb{R}^n$ comportant un point par cellule de niveau n.

Étant données les propriétés de l'opérateur de projection, il suffit d'étudier sur chaque droite $\overline{x} \times \mathbb{R}$ les polynômes *à une variable* $P(\overline{x}, X)$ où $P \in \mathcal{P}$ et $\overline{x}$ est un point de l'échantillon de niveau $n-1$.

En général les points des échantillons successifs que l'on va calculer seront des nombres algébriques. Au niveau 1, par exemple, l'algorithme d'isolation des racines va calculer une liste de points qui sont soit rationnels, soit codés par un intervalle $]a, b[$ et un polynôme T ayant une unique racine réelle dans l'intervalle $]a, b[$.

Ce codage des algébriques (éléments de $\hat{\mathbb{Q}}$) se généralise en dimensions supérieures aux éléments de $\hat{\mathbb{Q}}^n$ grâce aux *codages triangulaires.*

Définition 3.2.2 (Codages triangulaires) *Soit $z = (z_1, \ldots z_n)$ un élément de $\hat{\mathbb{Q}}^n$.*

- *Si $n = 1$, alors $z = z_1 \in \hat{\mathbb{Q}}$. Un codage triangulaire de z est*
  - *soit le singleton $\{z_1\}$ si $z_1 \in \mathbb{Q}$;*
  - *soit un couple (I_1, T_1) où $T_1 \in \mathbb{Q}[X]$ et I_1 un intervalle ouvert à bornes rationnelles tel que z_1 est l'unique racine de T_1 dans I_1.*
- *Si $n = m+1$, alors $z = (z_1, \ldots z_m, z_{m+1})$. Un codage triangulaire de z est un couple formé d'un codage triangulaire de $z' = (z_1, \ldots z_m)$ et de*
  - *soit le singleton $\{z_{m+1}\}$ si $z_{m+1} \in \mathbb{Q}$;*
  - *soit un couple (I_{m+1}, T_{m+1}) où $T_m + 1 \in \mathbb{Q}[X_1, \ldots, X_{m+1}]$ et I_{m+1} un intervalle ouvert à bornes rationnelles tel que z_{m+1} est l'unique racine du polynôme à* une *variable $T_{m+1}(z_1, \ldots, z_m, X_{m+1})$ dans I_{m+1}*

Remarque 3.2.3.1 *Il n'y a pas d'unicité du codage triangulaire pour un point : l'intervalle qui isole une racine peut être plus ou moins large, le polynôme annulateur n'est pas forcément minimal. Une coordonnée rationnelle peut être codée par sa propre valeur mais elle peut aussi être codée comme la racine d'un polynôme.*

A chaque niveau, chaque point des échantillons construit sera donné par un codage triangulaire. Au niveau 1, les éléments d'un échantillon construit à partir d'une liste d'isolation sont des codages algébriques d'après la définition 3.2.1. Puis on obtient récursivement les codages de points d'un échantillon de niveau n en construisant une liste d'isolation des racines réelles d'un polynôme $P(\overline{x}, X)$ où $P \in \mathcal{P}$ et $\overline{x}$ est un point de l'échantillon de niveau $n-1$.

Il reste toutefois à établir que l'on peut appliquer l'algorithme d'isolation des racines à un tel polynôme $P(\overline{x}, X)$.

Proposition 3.2.2 (Signe d'un algébrique) *Pour tout nombre algébrique $\overline{x} = (x_1, \ldots, x_k)$ de $\hat{\mathbb{Q}}^k$ donné par un codage triangulaire, il est possible de calculer le signe d'un élément de $\mathbb{Q}[x_1, \ldots, x_k]$.*

Preuve On effectue une récurrence sur la dimension k. Lorsque $k = 1$, le problème revient à calculer le signe pris par un polynôme $S \in \mathbb{Q}[X]$ en l'unique racine r d'un polynôme $T \in \mathbb{Q}[X]$ qui se trouve dans l'intervalle $]a, b[$ à bornes rationnelles. D'après la remarque 3.2.1.1, les algorithmes donnés en section 3.2.4 permettront de résoudre ce problème, puisqu'on sait calculer le signe d'un nombre rationnel.

Supposons qu' on sait calculer le signe des algébriques de $\hat{\mathbb{Q}}^k$ donnés par un codage triangulaire, et que $\overline{x} = (x_1, \ldots, x_{k+1}) \in \mathbb{Q}^{k+1}$. On veut calculer le signe de $P(\overline{x})$ où

$P \in \mathbb{Q}[X_1, \ldots, X_{k+1}]$. On peut mettre $\overline{x}$ sous la forme $T(x_{k+1})$ où $T \in \mathbb{Q}[x_1, \ldots, x_k]$. Cette transformation ne fait pas intervenir de calculs arithmétique sur les algébriques puisqu'il s'agit simplement d'identifier $\mathbb{Q}[X_1, \ldots, X_{k+1}]$ avec $\mathbb{Q}[X_1, \ldots, X][X_{k+1}]$, donc uniquement de manipuler les coefficients rationnels de P. Soit $((I_1, T_1), \ldots, (I_{k+1}, T_{k+1}))$ le codage de $\overline{x}$.

Le problème revient à calculer le signe de $T \in \mathbb{Q}[x_1, \ldots, x_k]$ en l'unique racine de $T_{k+1}(x_1, \ldots, x_k, X) \in \mathbb{Q}[x_1, \ldots, x_k][X]$ dans I_{k+1}. Toujours d'après la remarque 3.2.1.1, il suffit de montrer que l'on sait calculer le signe des éléments de $\mathbb{Q}[x_1, \ldots, x_k]$, ce qui est l'hypothèse de récurrence, et le signe d'un polynôme de $\mathbb{Q}[x_1, \ldots, x_k][X]$ évalué en un point rationnel. Or un polynôme de $\mathbb{Q}[x_1, \ldots, x_k][X]$ évalué en un point rationnel donne encore un élément de $\mathbb{Q}[x_1, \ldots, x_k]$, et le passage de l'une à l'autre des représentations ne fait intervenir que des calculs sur les nombres rationnels. On peut appliquer directement l'hypothèse de récurrence. □

Grâce à cette propriété, on s'assure des deux conditions de la remarque 3.2.1.1 et on pourra appliquer les algorithmes de la section 3.2.4.

Remarque sur la bonne formation des polynômes étudiés

Lors d'une étape de remontée, on s'intéresse à l'étude d'une famille de polynômes à $n+1$ variables, au dessus d'un point $\overline{\alpha}$ de $\mathbb{R}^n$ dont les coordonnées sont des algébriques. On étudie donc des polynômes de la forme $p_0(\overline{\alpha}) + \cdots + p_m(\overline{\alpha})X^n$ où $p_i \in \mathbb{Q}[X_1, \ldots, X_n]$.

Or pour la correction des calculs d'isolation de racines, et en particulier celui des bornes de Cauchy, il est important d'assurer que l'expression $p_0(\overline{\alpha}) + \cdots + p_m(\overline{\alpha})X^n$ décrit un polynôme bien formé, c'est à dire que $p_m(\overline{\alpha}) \neq 0$.

Une phase de remontée au dessus d'un point α de $\mathbb{R}^n$ d'une cellule de niveau n commence donc par le tri de la famille selon ce critère : pour chaque polynôme $P = p_0(X_1, \ldots, X_n) + \cdots + p_n(X_1, \ldots, X_n)X^{n+1}$ de la famille, on calcule sa forme non dégénérée au dessus du point α, c'est à dire une troncature $P_tronc = p_0(X_1, \ldots, X_n) + \cdots + p_m(X_1, \ldots, X_m)X^{m+1}$ telle que $p_m(\overline{\alpha}) \neq 0$ et pour tout $i = m+1 \ldots n$, $p_i(\overline{\alpha}) = 0$. Ce calcul de signe de la même façon que dans la preuve de la propriété 3.2.2.

Remarque sur le codage des nombres algébriques

Le codage utilisé dans cette implémentation ne fait pas partie des codages usuels employés pour représenter les nombres algébriques.

En effet, les bibliothèques d'arithmétique de nombres algébriques utilisent le plus souvent d'autres codages comme les représentations matricielles ou sous forme d'éléments des $\mathbb{Q}(\theta)$ pour un nombre algébrique θ fixé et connu.

La pertinence d'un choix de codage pour un développement manipulant des nombres algébriques dépend de deux facteurs : le premier est la façon dont les nombres algébriques apparaissent dans le problème (si ce ne sont pas des données initiales), le second est le type d'opérations que l'on doit effectuer sur ces nombres algébriques.

Dans l'algorithme de CAD les nombres algébriques avec lesquels on doit calculer apparaissent naturellement sous forme de codages triangulaires puisqu'ils sont obtenus par isolation de racines de polynômes. D'autre part, le type d'opération que l'on doit savoir calculer sur ces nombre est assez restreint : on ne fait jamais que des calculs de

signes de nombres algébriques. Plus précisément, on calcule le signe de polynômes à n variables évalués en un élément de R^n dont toutes les composantes sont des nombres algébriques, et qui est donné par un codage triangulaire.

Pour ceci, on a vu qu'il n'était jamais besoin d'implémenter une bibliothèque d'*arithmétique* sur les nombres algébriques, pour laquelle la représentation choisie n'est pas la plus efficace.

3.2.4 Isolation des racines réelles

Il s'agit maintenant d'expliciter les algorithmes qui permettent de construire une liste d'isolation pour une famille de polynômes ainsi que les signes des polynômes en les points d'un échantillon exhaustif construit à partir de cette famille.

On étudie une famille finie de polynômes à une variable $\mathcal{P} \subset K[X]$, où K est un sous-corps de $\mathbb{R}$ tel que les conditions nécessaires de la remarque 3.2.1.1 soient satisfaites.

La première étape consiste à se ramener à un problème sur un intervalle de longueur finie.

Proposition 3.2.3 (Borne de Cauchy) *Soit $P = a_pX^p + \cdots + a_qX^q \in K[X]$ avec $a_pa_q \neq 0$ et $p > q$.*

On définit :

$$C(P) := (p+1)\sum_{i=q}^{p} \frac{a_i^2}{a_p^2}$$

La valeur absolue de tout racine réelle de P est bornée par $C(P)$.

Preuve Soit $x \in \mathbb{R}$ une racine réelle de P. On a :

$$a_px = -\sum_{i=q}^{p-1} a_ix^{i-p+1}$$

Et :

$$(a_px)^2 = \left(\sum_{i=q}^{p-1} a_ix^{i-p+1}\right)^2$$

Comme $C(P) \geq 1$, on cherche un majorant pour les valeurs absolues des racines $|x| \geq 1$, pour lesquelles on a la relation suivante, due à la convexité de la fonction carré donne :

$$(a_px)^2 \leq (p+1)\sum_{i=q}^{p-1} a_i^2$$

D'où $|x| \leq C(P)$ pour toute racine réelle de P. □

Si l'utilisation de cette borne classique ne pose aucune difficulté pour des polynômes à une variable, quelques précisions supplémentaires sont nécessaires pour le cas des dimensions supérieures, où on manipule des coefficients algébriques. En effet, suivant la

remarques 3.2.1.1, nous n'avons jamais précisé que nous avions implémenté une arithmétique des nombres algébriques, au contraire, nous avons expliqué dans la preuve de la propriété 3.2.2 comment se ramener à des calculs rationnels tant qu'on ne s'intéresse qu'à des propriétés de signes.

Ici, nous ne sommes intéressés que par une borne des racines réelles, et en aucun cas par la valeur "exacte" de la quantité $C(P)$. Comme dans tous les cas qui nous occupent les coefficients a_i d'un polynôme sont donnés par un codage triangulaire, on dispose en fait d'un encadrement pour toutes les coordonnées de chaque coefficient d'un polynôme. Ainsi une arithmétique par intervalle grossière permettra de majorer la quantité $C(P)$ et d'obtenir la borne recherchée.

Il est à noter que la division par le coefficient dominant peut introduire des calculs supplémentaires. En effet pour appliquer cette arithmétique par intervalle, il est nécessaire d'isoler de zéro le nombre algébrique qui est le coefficient dominant du polynôme. Ceci peut demander de raffiner le codage triangulaire par dichotomie sur les intervalles, jusqu'à obtenir pour a_p^2 un encadrement disjoint de zéro. Nous reviendrons sur ce problème dans la section 4.2.3 du chapitre 4.

On peut supposer sans perdre de généralité que les polynômes dont on veut isoler les racines sont des polynômes à racines *simples*.

Définition 3.2.3 (Partie sans carré) *Soit $P \in D[X]$. On appelle* partie sans carrés *de P et on note $\overline{P}$ un pgcd de P et de P', dérivée du polynôme P'. Le polynôme $\overline{P}$ est à racines simples.*

On s'intéresse en fait aux racines réelles de polynômes et à leurs positions relatives mais pas à leur multiplicité.

Les informations sur les racines réelles d'un polynôme sur un intervalle borné donné vont être obtenues à partir des signes des coefficients du polynôme dans une base appropriée.

Définition 3.2.4 (Polynômes de Bernstein) *Soit $c, d \in \mathbb{Q}$ et $p \in \mathbb{N}$. La famille des polynômes de Bernstein de degré p est définie par :*

$$B_{p,i}(c,d) = \binom{p}{i} \frac{(X-c)^i (d-X)^{p-i}}{(d-c)^p} \quad 0 \leq i \leq p$$

Théorème 3.2.1 (Base de Bernstein) *Les polynômes de Bernstein de degré p pour les paramètres c, d forment une base de l'espace vectoriel des polynômes à une variable à coefficients dans un sur-corps de $\mathbb{Q}$ de degré inférieur ou égal p. Cette base est notée $B_p(c,d)$.*

Preuve On considère le polynôme $\dfrac{(X-c)^i (d-X)^{p-i}}{(d-c)^p}$. On lui applique successivement les deux transformations bijectives de l'ensemble des polynômes de $\mathbb{Q}[X]$ dans lui-même suivantes : $P(X) \mapsto P\left(\dfrac{X-c}{d-c}\right)$ suivie de $P(X) \mapsto X^p P(1/X)$. On obtient alors le polynôme $(X-1)^{p-i}$. □

La propriété des bases de Bernstein qui nous intéresse ici est que les coefficients d'un polynôme quelconque dans une de ces bases donne suffisamment d'information pour savoir quand un intervalle contient une unique racine de ce polynôme.

Définition 3.2.5 (Nombre de changement des signes) *Soit $a := a_0, \ldots a_n$ une suite finie de nombres réels. Le nombre de changements de signes $ncs(a)$ dans cette suite est défini par récurrence sur la longueur de la liste, comme suit :*

- *Si $a := a_0$, avec $a_0 \in \mathbb{R}$ un réel quelconque, alors $V_a := 0$.*
- *Si $a := a', a_n, a_{n+1}$ où a' est une suite finie de nombres réels qui peut être vide, alors*
  - *Si $a_{n+1} = 0$, alors $V_a := ncs(a', a_n)$*
  - *Si $a_n = 0$, alors $V_a := ncs(a', a_{n+1})$*
  - *Sinon,*
    - *Si $a_{n+1}a_n > 0$, alors $V_a := ncs(a', a_n)$*
    - *Si $a_{n+1}a_n < 0$, alors $V_a := ncs(a', a_n) + 1$*

Théorème 3.2.2 *Soit $P \in K[X]$ de degré p, à racines simples, et $c < d \in \mathbb{Q}$.Soit $b = (b_i)_{i=0\ldots p}$, la suite des coordonnées b_i de P sur $B_{p,i}(c,d)$ dans la base de Bernstein $B_p(c,d)$. On note $V_b(P)$ le nombre de changements de signes dans la suite b. On a alors :*

- *$V_b(P) = 0$ si et seulement si P n'a pas de racine sur $]c,d[$*
- *$V_b(P) = 1$ si et seulement si P a exactement une racine dans l'intervalle $]c,d[$.*

Dans les autres cas, on ne possède pas une information suffisante pour tirer une conclusion sur le nombre de racines réelles de P dans $]c,d[$.

Cette information partielle est néanmoins suffisante car le caractère archimédien de $\mathbb{R}$ assure qu'un procédé de dichotomie amènera immanquablement à des intervalles de longueur assez petite pour que l'on se trouve dans l'une ou l'autre des deux situation de théorème 3.2.2

Cet oracle qui permet de tester un intervalle candidat à devenir élément d'une liste d'isolation est suffisant pour résoudre le problème complet de l'isolation des racines d'une famille et le calcul des signes réalisés par un échantillon exhaustif. Si le test échoue, c'est à dire si le nombre de changement de signes de la liste de coefficients est strictement plus grand que 1, alors on va initier un processus de dichotomie, en testant le milieu de l'intervalle candidat ainsi que ses deux moitiés ouvertes.

Remarque 3.2.4.1 *Pour la terminaison de ces procédés dichotomiques, le fait de travailler dans un corps réel clos archimédien est fondamental. Le traitement des corps réels non archimédien se fait à l'aide de codages à la Thom (utilisant la liste des signes des dérivées successives).*

On détaille ici le calcul d'une liste d'isolation pour une famille réduite à deux polynômes P_1 et P_2. La généralisation à une famille finie quelconque ne pose pas de difficulté. On note $Bern(P,a,b)$ l'appel à la fonction qui calcule le nombre de changement de signes de la liste des coefficients du polynôme P dans la base de Bernstein de degré $deg(P)$ pour les paramètres a et b.

Pour pouvoir distinguer les racines communes de P_1 et P_2 des intervalles encore trop grossiers qui contiennent une racine de P_1 et une racine distincte de P_2, on aura besoin d'utiliser le pgcd de P_1 et P_2. Dans le cas de base des polynômes à coefficients rationnels,

l'algorithme d'Euclide est suffisant pour ce calcul, dans le cas des polynômes à coefficients algébriques, on calcule en fait des pgcd de polynômes dans $\mathbb{Q}[X_1, \ldots, X_k][Y]$, c'est à dire à coefficients dans un anneau. On utilisera alors l'algorithme des sous-résultants du chapitre 2.

Tout d'abord, l'algorithme de la figure 3.12 permet de calculer le signe du polynôme P_2 en l'unique racine de P_1 contenue dans l'intervalle $]c, d[$ à bornes rationnelles.

Comme ce calcul peut comporter un raffinement de l'encadrement de l'intervalle qui isole la racine de P_2, on retourne à la fois le signe deP_1 et un nouvel encadrement pour la racine de P_2 (qui est un sous-intervalle de $]c, d[$).

Initialisation
- $P_1, P_2 \in D[X]$ sont des polynômes à racines simples.
- P_2 a une unique racine dans l'intervalle $]c, d[$ à bornes rationnelles.

Procédure
- si $Bern(P_2, c, d) = 0$, alors évaluer P_2 en $\frac{c+d}{2}$:

  rendre $signe(P_2(\frac{c+d}{2}))$ et $]c, d[$
- si $Bern(P_2, c, d) = 1$ alors on doit décider si P_1 et P_2 ont une racine commune entre c et d ou non. Soit G le pgcd de P_1 et P_2.
  - si $Bern(\overline{G}, c, d) = 1$, alors la racine est commune :

    rendre 0 et $]c, d[$.
  - si $Bern(\overline{G}, c, d) = 0$, alors raffiner $]c, d[$ par dichotomie pour obtenir un sous-intervalle (ou un singleton) u' de $]c, d[$ qui ne contienne plus la racine de P_1 mais contient celle de P_2. L'intervalle u' qui convient est le premier tel que la suite des coefficients de Bernstein de $\overline{G}$ n'ait plus de changement de signe. Évaluer P_1 en le milieu $mid(u')$ de u'.

    rendre $signe(P_2(mid(u')))$ et u'.
  - si $Bern(\overline{G}, c, d) > 0$, alors raffiner $]c, d[$ par dichotomie, pour obtenir un intervalle (ou un singleton) qui relève de l'un des cas précédents. À chaque étape, il faut tester chacun des demi-intervalles ouverts ainsi que le point médian.

FIG. 3.12: *Signe d'un polynôme en une racine isolée d'un autre*

Et maintenant, on peut calculer une liste d'isolation puis un échantillon exhaustif, ainsi que toutes les conditions de signes réalisées simultanément par les polynômes P_1 et P_2, grâce à l'algorithme de la figure 3.13.

Enfin il nous reste à préciser la façon dont les coefficients de Bernstein sont calculés au cours de l'algorithme. Retrouver à chaque dichotomie les coefficients du polynôme par les transformations de la propriété fondamentale 3.2.1 serait coûteux. En fait il existe une relation du type *triangle de Pascal* entre les coefficients de Bernstein d'un polynôme donné pour les paramètres g, d et les coefficients du même polynôme, dans les bases de même degré mais de paramètres respectifs g, m et m, d. Il en découle un algorithme quadratique en le degré du polynôme.

Proposition 3.2.4 (Calcul des coefficients de Bernstein) *Soit P un polynôme dans $K[X]$ de degré $\leq p$, et b la liste de ses coefficients dans la base de Bernstein de degré p et*

Initialisation
$P_1, P_2 \in D[X]$ sont des polynômes à racines simples.

Procédure
- Calculer les bornes des racines de P_1 et P_2, et prendre les extrémas A et B qui encadrent les racines de la famille
- Calculer une liste d'isolation L_{P_1} pour le polynôme P_1 sur $]A, B[$ en utilisant l'algorithme suivant :
- Pour calculer une liste L_{P_1} pour P_1 sur un intervalle ouvert $]a, b[$:
  - si $Bern(P_1, a, b) = 0$ alors $L_{P_1} = \{\}$
  - si $Bern(P_1, a, b) = 1$ alors $L_{P_1} = \{]a, b[\}$
  - sinon
    - calculer L'_{P_1} une liste d'isolation de P_1 sur $]a, \frac{a+b}{2}[$,
    - calculer L''_{P_1} une liste d'isolation de P_1 sur $]\frac{a+b}{2}, b[$,
    - si $P_1(\frac{a+b}{2}) = 0$ alors $L_{P_1} = L'_{P_1} \cup L''_{P_1} \cup \{[\frac{a+b}{2}]\}$
    - sinon $L_{P_1} = L'_{P_1} \cup L''_{P_1}$
- si $L_{P_1} = \{\}$ alors calculer une liste d'isolation pour P_2 sur $]a, b[$
- sinon pour chaque élément u de L_{P_1}, calculer le signe de P_2 sur u :
  - si u est un singleton rationnel, alors évaluer P_2
  - si $u =]c, d[$ alors calculer le signe de P_2 en l'unique racine de P_1 contenue dans $]c, d[$.

  À présent, nous avons construit une liste d'isolation L_{P_1} pour P_1, telle que chaque élément de L_{P_1} est une racine de P_2 ou bien un intervalle sur lequel P_2 a un signe constant, qui a été calculé.
- Pour chaque intervalle entre deux éléments consécutifs de L_{P_1}, calculer une liste d'isolation pour P_2 comme ci-dessus, et calculer le signe de P_1. Fusionner cette liste avec L_{P_1} pour obtenir L_{P_1,P_2}, une liste d'isolation pour la famille $\{P_1, P_2\}$.
- Calculer le signe de P_1 et P_2 en A et B
- Calculer le signe de P_1 et P_2 entre deux éléments consécutifs de L_{P_1,P_2}, en évaluant les polynômes en le milieu des bornes des intervalles. Si $L_{P_1,P_2} = \emptyset$, évaluer les polynômes en 0.

FIG. 3.13: *Calcul d'une liste d'un échantillon exhaustif signé pour une famille de deux polynômes*

de paramètres $g, d \in \mathbb{Q}$. Soit $m \in \mathbb{Q}$. Alors, les coefficients b' et b'' de P dans les bases de Bernstein de degré p respectivement pour les paramètres g, m et m, d sont donnés par :

$$b' = b_0^0 \dots b_0^j \dots b_0^p$$
$$b'' = b_0^p \dots b_j^{p-j} \dots b_p^0$$

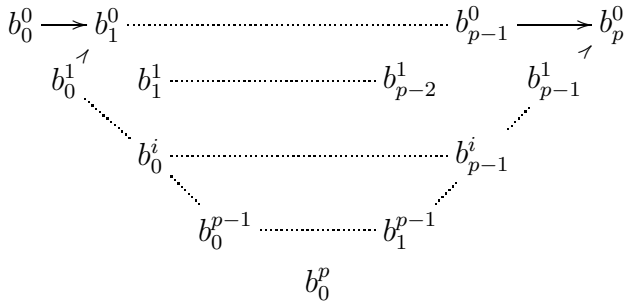
où b_j^i est défini par :

$$b_0^j = b_j \quad et \quad b_j^i = \alpha b_j^{i-1} + \beta b_{j+1}^{i-1}$$

avec $\alpha = \dfrac{d-m}{d-g}$ *et* $\beta = \dfrac{m-l}{d-g}$.

Voir la preuve de correction de l'algorithme 10.2, page 365 dans (Basu, Pollack, et Roy 2006).

En fait, la structure de triangle de Pascal sera plus claire si on visualise le diagramme des b_j^i :



C'est le côté gauche du triangle qui constitue l'initialisation de l'algorithme. Le but est de calculer les coefficients des deux autres côtés du triangle, sur lesquels on lit les coefficients des listes b' et b''. Pour cela on va en fait devoir calculer tous les b_i^j,rangée par rangée, en partant de la plus à gauche qui est formée des coefficients de b. Pour cela, on utilise la relation dans le sens

$$b_{j+1}^{i-1} = \frac{b_j^i - \alpha b_j^{i-1}}{\beta}$$

Il existe plusieurs variantes des algorithmes d'isolation des racines réelles à l'aide d'arguments issus de la règle des signes de Descartes. Ces algorithmes sont souvent désignés par le terme générique de *méthode d'Uspensky* (Uspensky 1948), dénomination relativement impropre puisqu'ils apparaissent de façon originale dans des travaux antérieurs (Vincent 1836).

Ces procédés dichotomiques sont propres au corps des réels. Pour traiter le cas des corps réels clos non archimédiens, il faut utiliser d'autres méthodes, par exemple basées sur les suites de Sturm.

Par contre, les méthode *à la* Uspensky conduisent à des algorithmes efficaces (Rouillier et Zimmermann 2004) dès que l'on travaille avec des nombres réels. En particulier l'utilisation des polynômes de Bernstein permet de calculer efficacement des approximation de racines réelles, même en dimension supérieure (Mourrain, Rouillier, et Roy 2005), et ces algorithmes sont implémentés et utilisés dans la librairie de calculs symboliques et numériques SYNAPS (Dos Reis, Mourrain, Rouillier, et Trébuchet 2002).

On propose un exemple de calcul d'une liste d'isolation dans le cas de polynômes à une variable dans la section 3.3.1.

3.2.5 Opérateur de projection

La dernière étape qu'il nous reste à décrire est l'opérateur de projection. Il faut trouver une transformation qui assure les propriétés de la proposition 3.2.1. Plus précisément, étant donnée une famille finie $\mathcal{P} \subset \mathbb{Q}[X_1, \ldots, X_n][X_{n+1}]$ de polynômes, il faut trouver un moyen systématique de calculer une famille $\mathcal{Q} \subset \mathbb{Q}[X_1, \ldots, X_n]$, dans laquelle on a éliminé la variable X_{n+1}, telle que l'invariance des signes de tous les polynômes de $\mathcal{Q}$ sur un semi-algébrique $S \subset \mathbb{R}^n$ assure que :

- pour tout polynôme $P \in \mathcal{P}$ le degré et le nombre de racines du polynôme $P(\overline{x}, X_{n+1})$ est indépendant du choix de $\overline{x}$ dans S ;
- pour tous $P_1, P_2 \in \mathcal{P}$, le degré du pgcd de $P_1(\overline{x}, X_{n+1})$, et $P_2(\overline{x}, X_{n+1})$ est indépendant du choix de $\overline{x}$ dans S.

Dans la suite, on note $D_k[X]$ l'espace vectoriel des polynômes de degré strictement inférieur à k. Dans les représentations matricielles des morphismes d'espaces vectoriels, on identifie un couple de polynômes $(U, V) \in D_k[X] \times D_l[X]$ au $k+l$-uplet $(u_k, \ldots, u_0, v_l, \ldots, v_0)$ des coordonnés des polynômes U et V dans les bases monomiales concaténées.

Assurer les conditions sur les degrés des pgcds

C'est l'utilisation des *coefficients sous-résultants* qui va permettre de définir un tel opérateur de correction en limitant l'explosion combinatoire dans la famille projetée. L'efficacité de cet opérateur de projection est le point central de la différence de complexité entre l'algorithme de CAD et ses prédécesseurs historiques.

Soit D un anneau factoriel, on considère deux polynômes $P = p_n X^n + \cdots + p_0$, tel que $p_n \neq 0$ et $Q = q_m X^m + \cdots + q_0$ tel que $q_m \neq 0$ deux polynômes de $D[X]$. On définit $f_{P,Q}$, application linéaire associée à ces deux polynômes par :

$$\begin{array}{rcl} f_{P,Q} : D_{m-1}[X] \times D_{n-1}[X] & \longrightarrow & D_{n+m-1}[X] \\ (U, V) & \longrightarrow & UP + VQ \end{array}$$

Définition 3.2.6 (matrice de Sylvester, résultant) *La matrice $M_{P,Q}$ de taille $(n+m) \times (n+m)$ de l'application linéaire $f_{P,Q}$ est*

$$\mathbf{M}_{P,Q} = \begin{bmatrix} p_n & & 0 & q_m & & 0 \\ \vdots & \ddots & & \vdots & \ddots & \\ \vdots & & p_n & & & q_m \\ & & & q_0 & & \vdots \\ p_0 & & \vdots & \vdots & \ddots & \\ & \ddots & \vdots & \vdots & & \\ 0 & \ldots & p_0 & 0 & \ldots & q_0 \end{bmatrix}$$

Sa transposée est appelée matrice de Sylvester de P et Q.

$$Sylv(P,Q) = \begin{bmatrix} p_n & p_{n-1} & p_{n-2} & \cdots & \cdots & \cdots & \cdots & p_0 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & p_n & p_{n-1} & p_{n-2} & \cdots & \cdots & \cdots & \cdots & p_0 \\ q_m & q_{m-1} & q_{m-2} & \cdots & \cdots & \cdots & q_0 & 0 & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & q_m & q_{m-1} & q_{m-2} & \cdots & \cdots & \cdots & q_0 \end{bmatrix}$$

Le déterminant de la matrice de Sylvester de P et Q (qui est aussi celui de $M_{P,Q}$) est appelé résultant de P et Q et noté $Res(P,Q)$.

La propriété essentielle du résultant $Res(P,Q)$ de deux polynômes le relie au degré de $pgcd(P,Q)$.

Proposition 3.2.5 *$Res(P,Q) = 0$ si et seulement si P et Q ont un facteur commun.*

Voir Proposition 4.13 dans (Basu, Pollack, et Roy 2006).

Les restrictions de $f_{P,Q}$ aux espaces vectoriels des polynômes de degré inférieur se relient également au pgcd. Supposons que $m < n$. Pour tout $0 \leq j \leq m$ on définit :

$$\begin{array}{rcl} f_{P,Q,j} : D_{m-j}[X] \times D_{n-j}[X] & \longrightarrow & D_{n+m-2j-2}[X] \\ (U,V) & \longrightarrow & UP + VQ \end{array}$$

Définition 3.2.7 (matrice de Sylvester-Habicht, coefficients sous-resultants) *La matrice $SH_j(P,Q)$ de l'application $f_{P,Q,j}$ dans les bases monomiales concaténées de $D_{m-j}[X] \times D_{n-j}[X]$ est appelée $j^{ème}$ matrice de Sylvester-Habicht de P et Q.*

Le $j^{ème}$ coefficient sous-résultant de P et Q, noté $sr_j(P,Q)$, est le déterminant de la matrice carrée obtenue en prenant les $n_m - 2j$ premières colonnes de $SH_j(P,Q)$.

$$\mathbf{SH}_j(P,Q) = \begin{bmatrix} p_n & p_{n-1} & p_{n-2} & \cdots & \cdots & \cdots & \cdots & p_0 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & p_n & p_{n-1} & p_{n-2} & \cdots & \cdots & \cdots & \cdots & p_0 \\ q_m & q_{m-1} & q_{m-2} & \cdots & \cdots & \cdots & q_0 & 0 & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & q_m & q_{m-1} & q_{m-2} & \cdots & \cdots & \cdots & q_0 \end{bmatrix}$$

On étend cette définition en posant par convention :

$$sr_n(P,Q) = signe(p_n)$$
$$sr_j(P,Q) = 0 \quad pour \quad m < j < n$$

On peut maintenant raffiner la propriété 3.2.5 :

Proposition 3.2.6 *Pour tout* $0 \leq j \leq min(n,m)$ *(resp.* $0 \leq j \leq n-1$ *if* $n = m$*),*

$$d^o(pgcd(P,Q)) \geq j$$

$$\Leftrightarrow$$

$$sr_0(P,Q) = \cdots = sr_{j-1}(P,Q) = 0$$

Voir Proposition 4.24 dans (Basu, Pollack, et Roy 2006).

On rappelle que les coefficients sous-résultants sont les coefficients dominants des polynômes sous-résultants (voir 2.4.1), ce qui fournit un moyen de les calculer plus efficacement que par les mineurs de la matrice de Sylvester.

Assurer les conditions sur les degrés des polynômes

La première condition de la proposition 3.2.1 demande que les polynômes restent de la même forme sur une même cellule, c'est à dire qu'on ait pas d'annulation possible dans les coefficients dominants des polynômes de la famille à étudier.

Pour assurer cette propriété, on va compléter la famille $\mathcal{P}$ qui fait l'objet de l'étude par l'ensemble de ses *troncatures*.

Définition 3.2.8 (Troncatures) *Soit* D *un anneau intègre et* $P = p_nX^n + \cdots + p_0 \in D[X]$*. Pour* $0 \leq i \leq n$*, on définit la* troncature de degré i *du polynôme* P *comme :*

$$Tronc_i(P) = p_iX^i + \cdots + p_0.$$

Si $P \in D[Y_1, \ldots, Y_k][X]$*, est un polynôme non nul, on définit* l'ensemble des troncatures $Tronc(P)$ *du polynôme* P *par :*

– $Tronc(P) = \{P\}$ *si* $deg(P) = 0$ *ou bien* $coef\_dom(P) \in D$

– $Tronc(P) = \{P\} \cup Tronc(Tronc_{deg(P)-1})$ *sinon*

paral/

où $deg(P)$ *est le degré de du polynôme* P *(en la variable* X*) et* $coef\_dom(P)$ *est le coefficient dominant du polynôme* P *(en la variable* X*).*

Si $\mathcal{P} \subset D[Y_1, \ldots, Y_k][X]$ *est une famille de polynôme, par définition :*

$$Tronc(\mathcal{P}) = \bigcup_{P \in \mathcal{P}} Tronc(P)$$

Par exemple, si dans la famille $\mathcal{P}$ qu'on veut projeter se trouve le polynôme $X^2Y^3 + (X+1)Y^2 + Y + 1$, on doit compléter la famille avec les polynômes $(X+1)Y^2 + Y + 1$ et $Y + 1$. Par contre, il n'est pas nécessaire de rajouter à la famille le polynôme constant 1 puisque le coefficient dominant de $Y + 1$ ne peut pas s'annuler.

Définition de l'opérateur de projection

On définit l'*opérateur d'élimination par sous-résultants* qui réunit les outils ce-dessus.

Définition 3.2.9 (Élimination par sous-résultants) *Soit $\mathcal{P} \in \mathbb{Q}[X_1, \ldots, X_k][X_{k+1}]$ une famille finie de polynômes. On définit $Elim_{X_{k+1}} \subset \mathbb{Q}[X_1, \ldots, X_k]$ par :*

- *si $R \in Tronc(\mathcal{P})$, alors $Elim_{X_{k+1}}$ contient tous les $sr_i(R, \partial R/\partial X_k)$ qui ne sont pas dans $\mathbb{Q}$, pour $i = 0 \ldots deg_{X_k}(R) - 2$;*
- *pour tous $R, S \in Tronc(\mathcal{P})$,*
  - *si $deg_{X_k}(R) < deg_{X_k}(S)$ alors $Elim_{X_{k+1}}$ contient tous les $sr_i(S, R)$ qui ne sont pas dans $\mathbb{Q}$, pour $i = 0 \ldots deg_{X_k}(R) - 1$;*
  - *si $deg_{X_k}(S) < deg_{X_k}(R)$ alors $Elim_{X_{k+1}}$ contient tous les $sr_i(R, S)$ qui ne sont pas dans $\mathbb{Q}$, pour $i = 0 \ldots deg_{X_k}(S) - 1$;*
  - *si $deg_{X_k}(R) = deg_{X_k}(S)$ alors $Elim_{X_{k+1}}$ contient tous les $sr_i(S, \overline{R})$ qui ne sont pas dans $\mathbb{Q}$, où $\overline{R} = coef\_dom_{X_k}(S)R - coef\_dom_{X_k}(R)S$, pour $i = 0 \ldots deg_{X_k}(\overline{R}) - 1$;*
- *si $R \in Tronc(\mathcal{P})$, et si $coef\_dom(R) \notin \mathbb{Q}$ alors $coef\_dom(R) \in Elim_{X_{k+1}}$.*

Sur les cellules d'une partition adaptée à $Elim_{X_{k+1}}(\mathcal{P})$, les polynômes ont un degré constant, puisque leurs coefficients dominants *consécutifs* ne s'annulent pas. Ils ont un nombre de racines constant puisque par continuité des racines, un changement du nombre de racines se traduirait par un changement de multiplicité. Or le pgcd des polynômes avec leurs dérivées respectives ne changent pas de degré vue la propriété 3.2.6 et la première condition de la définition de $Elim_{X_{k+1}}(\mathcal{P})$. Enfin, cette même propriété 3.2.6 assure que sur les cellules adaptées à $Elim_{X_{k+1}}(\mathcal{P})$, le pgcd de deux polynômes de $(\mathcal{P})$ ne changent pas de degré puisque $Elim_{X_{k+1}}(\mathcal{P})$ contient les coefficients sous-résultants de tout couple d'éléments de $(\mathcal{P})$.

Nous avons tous les ingrédients pour construire l'algorithme qui calcule une décomposition algébrique cylindrique pour toute famille de polynômes à plusieurs variables et à coefficients rationnels.

Théorème 3.2.3 (Existence d'une CAD) *Pour tout entier $k \geq 1$, pour toute famille finie de polynômes de $\mathcal{P} \subset \mathbb{Q}[X_1, \ldots, X_k]$, on sait calculer une décomposition algébrique cylindrique adaptée à la famille $\mathcal{P}$.*

Les algorithmes présentés permettent également de construire un *échantillon cylindrique de points* issus de la décomposition algébrique cylindrique calculée pour une famille de polynômes arbitraire.

3.3 Exemples de calculs

Cette section est consacrée à des exemples de calculs illustrant les algorithmes décrits dans les premières parties de ce chapitre.

3.3.1 Isolation des racines par les polynômes de Bernstein

Dans cette section, on détaille la construction d'une *liste d'isolation* pour les polynômes de la famille $\{P_1 := (X-2)(X-3), P_2 := (X-1)(X-3)\}$. Les deux polynômes qui la constituent ont une racine commune et chacun a une deuxième racine, qui lui est propre.

Avec les propriétés des polynômes de Bernstein citées dans la section 3.2.4, on peut disposer d'un oracle *test* qui prend en argument un polynôme $P \in \mathbb{Q}[X]$ et deux rationnels $c, d \in \mathbb{Q}$ et qui renvoie trois types de réponses :

- $test(P, c, d) = 0 \quad \Leftrightarrow \quad$ il n'y a pas de racine pour P dans $]c, d[$
- $test(P, c, d) = 1 \quad \Leftrightarrow \quad$ il y a exactement une racine pour P dans $]c, d[$
- $test(P, c, d) = \perp \quad \Rightarrow \quad$ dans les autres cas, l'oracle échoue.

On réalise cet oracle en calculant les coefficients de Bernstein du polynôme P en argument sur l'intervalle ouvert $]c, d[$ et en comptant le nombre de changements de signes dans la suite obtenue.

On commence par calculer un majorant des bornes des Cauchy des deux polynômes, à l'aide de la formule de la propriété 3.2.3. Le calcul donne $max(C(P_1), C(P_2)) = 186$

Liste d'isolation pour P_1

Les calculs se déroulent de la façon suivante :

- On teste les deux sous-intervalles $]-186, 0[$ et $]0, 186[$ avec l'oracle *test*, pou le polynômes P_1.
- Le polynôme P_1 ne s'annule pas en 0.
- Le polynôme P_1 n'a pas de racine sur $]-186, 0[$.
- Sur $]0, 186[$, l'oracle ne sait pas répondre : il faut continuer le procédé de dichotomie.
- On teste chaque moitié de l'intervalle $]0, 186[$ grâce à l'oracle *test* et on évalue P_1 en 93. Il n'y a pas de racine sur $]93, 186[$, ni en 93. On réitère le découpage sur $]0, 93[$.
- L'étape précédente (découpage, test du milieu, conservation de la partie gauche) se répète jusqu'à ce que l'intervalle de travaille soit $]0, \frac{93}{8}[$.
- P_1 n'a pas de racine en $\frac{93}{8}$, mais l'oracle *test* permet d'identifier une racine dans chacun des sous-intervalles $]0, \frac{93}{16}[$ et $]\frac{93}{16}, \frac{93}{8}[$.

On a donc construit un liste d'isolation pour le polynôme P_1, illustrée sur la figure 3.14 :

$$\{]0, \frac{93}{16}[,]\frac{93}{16}, \frac{93}{8}[\}$$

Ce travail sur le polynôme P_1 divise l'intervalle d'étude $]-186, 186[$ en 4 sous-intervalles sur lesquels on va isoler les racines du polynôme P_2 : $]-\infty, 0[,]0, \frac{93}{16}[,]\frac{93}{16}, \frac{93}{8}[$ et $]\frac{93}{8}, +\infty[$.

Liste d'isolation pour P_2

- Le polynôme P_2 n'a pas de racine sur $]-\infty, 0[$, ni sur $]\frac{93}{32}, +\infty[$.

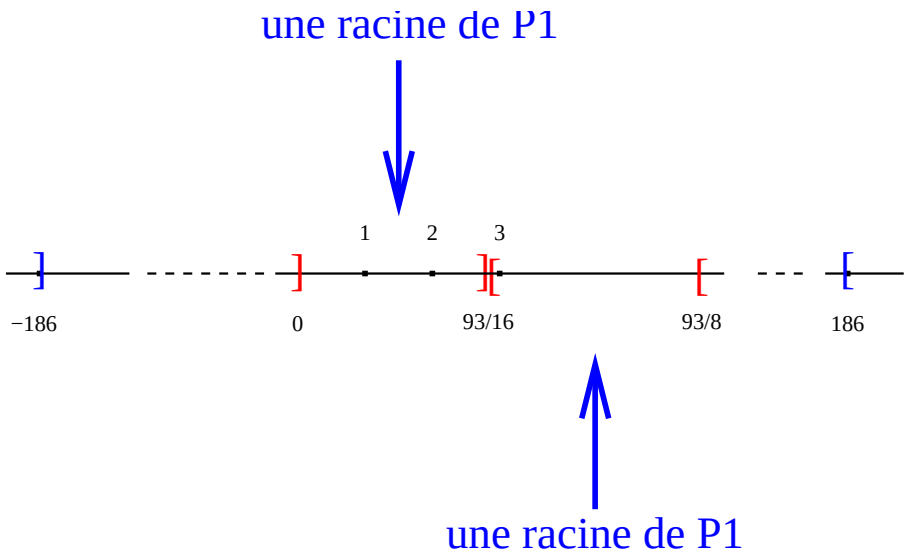


FIG. 3.14: *Isolation des racines de* $(X-1)(X-2)$

- Le polynôme P_2 a une unique racine sur chacun des intervalles $]0, \frac{93}{16}[$ et $]\frac{93}{16}, \frac{93}{8}[$.
- Il s'agit de déterminer si chacune de ces racines est commune avec la racine du polynôme P_1 qui se trouve aussi dans l'intervalle.

Discrimination des racines de P_1 et P_2

- On calcule $G := P_1 \wedge P_2 = X - 2$ le pgcd de P_1 et P_2.
- On interroge l'oracle *test* sur chacun des intervalles $]0, \frac{93}{16}[$ et $]\frac{93}{16}, \frac{93}{8}[$.
- G a une racine sur $]\frac{93}{16}, \frac{93}{8}[$, mais pas sur $]0, \frac{93}{16}[$, comme représenté en figure 3.15.
- Par suite, l'intervalle $]\frac{93}{16}, \frac{93}{8}[$ contient une racine commune à P_1 et P_2 et peut être placé dans la liste d'isolation de la famille $\{P_1, P_2\}$. Par contre, l'intervalle $]0, \frac{93}{16}[$ doit être raffiné car il contient deux racines distinctes, l'une de P_1 et l'autre de P_2, qu'il faut isoler.
- On initie un procédé de dichotomie sur l'intervalle $]0, \frac{93}{16}[$. Pour tester chaque sous-intervalle, on interroge l'oracle *test* sur chacun des polynômes P_1 et P_2.
- Dans ce cas, il faut deux pas de dichotomie : $]0, \frac{93}{64}[$ isole la racine de P_2 et $]\frac{93}{64}, \frac{93}{32}[$, celle de P_1

Finalement, la liste d'isolation construite est :

$$\{]0, \frac{93}{64}[,]\frac{93}{64}, \frac{93}{32}[,]\frac{93}{16}, \frac{93}{8}[\}$$

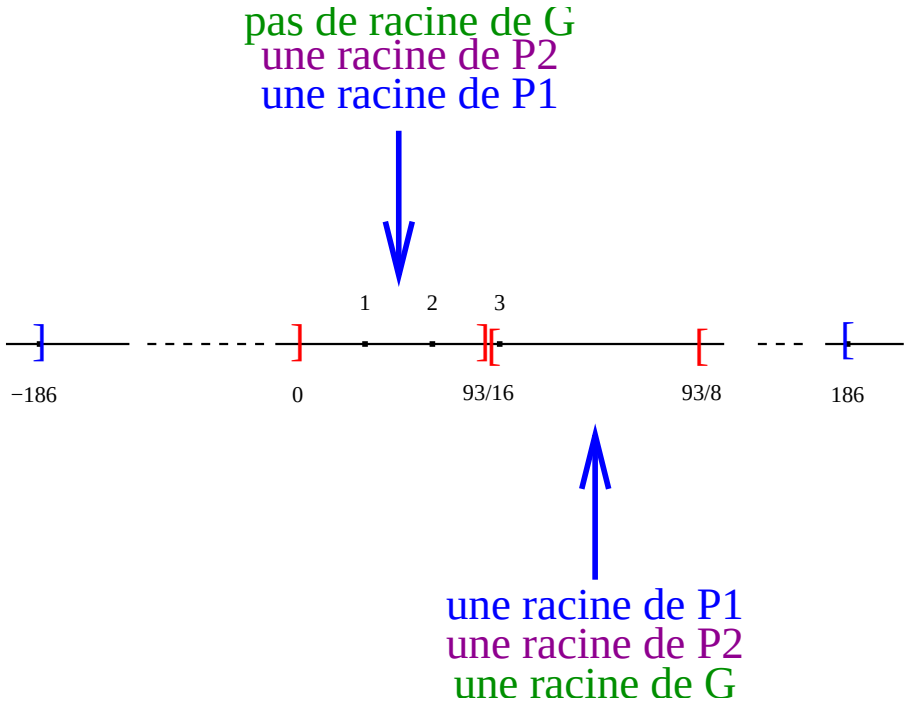


FIG. 3.15: *Discrimination des racines communes de* $\{(X+1)(X-2),(X-1)(X-3)\}$

3.3.2 Raffinement des codages des algébriques, arithmétique par intervalle

L'utilisation de bornes de Cauchy (voir propriété 3.2.3) dans la première étape du procédé d'isolation de racines réelles nécessite un peu de travail pour le cas des polynômes à une variable mais à coefficients algébriques.

En effet supposons qu'on utilise cette formule pour isoler les racines d'un polynômes $P(\overline{\alpha},X)$, avec $\overline{\alpha}+(\alpha_1,\ldots,\alpha_n)$ donné par un codage triangulaire (voir la définition 3.2.2), et $P = p_0+\cdots+p_mX^n$, avec $p_i \in \mathbb{Q}[X_1,\ldots,X_n]$. On veut donc isoler les racines du polynôme à une variable $p_0(\alpha)+\cdots+p_m(\alpha)X^n$.

Notre but étant de borner les racines réelles du polynôme $P(\overline{\alpha},X)$, on utilise la formule de Cauchy qui donne une borne sous la forme d'une fraction rationnelle en les composantes algébriques α_i de l'élément α, dont le dénominateur est une puissance de $p_m(\alpha)$.

Comme on dispose pour chacun des α_i d'un encadrement par un intervalle à bornes rationnelles, on va en fait utiliser une *arithmétique par intervalle grossière* pour obtenir une majoration de $C(P(\overline{\alpha}))$.

Rappelons (voir la remarque sur la bonne formation des polynômes de la section 3.2.3) que les polynômes pour lesquels on initie un calcul de borne de Cauchy sont bien formés, c'est à dire qu'on a calculé le signe de leur coefficient dominant, qui n'est pas zéro.

Cette approximation doit en particulier nous garantir que la quantité $p_m(\alpha)$ est séparée de zéro. Or, même si le calcul de signe d'un algébrique donne un signe non nul pour la quantité $p_m(\alpha)$, l'approximation rationnelle des coordonnées algébriques du point α peut ne pas être assez fine pour isoler cette variable de zéro.

Voici un exemple d'une telle situation.

Supposons que une phase de remontée conduise à étudier les racines réelles du poly-

nôme $P := (\alpha - 2)Y + 1$, où α est un point d'une cellule de niveau 1, c'est à dire un nombre réel algébrique.

Supposons que α est codé comme l'unique racine du polynôme $X - 1$ dans l'intervalle $]0, 3[$.

On applique donc la formule de Cauchy pour borner l'intervalle contenant les racines de P :

$$C(P) = (1 + 1)\frac{(\alpha - 2)^2 + 1}{(\alpha - 2)^2}$$

On veut déterminer un majorant pour la quantité $C(P)$, à l'aide de l'encadrement que l'on connaît pour α.

- Il faut tout d'abord raffiner l'encadrement puisque $0 < \alpha < 3$, soit $-2 < \alpha - 2 < 1$, ne permet pas d'isoler le numérateur de 0.
- On initie donc un procédé de dichotomie sur l'intervalle $]0, 3[$: on teste si $\frac{3}{2}$ est racine de $X - 1$ puis on calcule les coefficients de Bernstein du polynôme $X - 1$ sur chacun des demi-intervalles. On raffine ainsi l'encadrement, en obtenant $\alpha \in]0, \frac{3}{2}[$.
- Cette fois, l'encadrement est suffisamment précis : $-2 < \alpha - 2 < -\frac{1}{2}$, d'où $C(P) \leq 10$.

L'encadrement est grossier, mais comme les racines sont isolées par dichotomie, le procédé de souffre pas de cette imprécision.

3.3.3 Calcul de la CAD du cercle

On s'intéresse ici au calcul de la décomposition algébrique cylindrique de la famille $\mathcal{P}$ constituée d'un seul polynôme $P := X^2 + Y^2 - 1 \in \mathbb{Q}[X, Y]$, dont l'ensemble des racines est le cercle unité.

Phase d'élimination

D'abord, on calcule successivement les familles de polynômes de la phase d'élimination. Il n'y a que deux variables, supposons que l'on élimine Y en premier.

- $\mathcal{C}_2(\mathcal{P}) = \mathcal{P}\{X^2 + Y^2 - 1\}$
- $\mathcal{C}_1(\mathcal{P}) = Elim_Y(\mathcal{P}) = \{X^2 - 1\}$

La famille $\mathcal{C}_1$ est également réduite à un unique polynôme car il n'y a pas de troncature à considérer, et un seul coefficient sous-résultant à calculer.

Phase de remontée

La phase de remontée commence par l'isolation des racines réelles des polynômes de la famille $\mathcal{C}_1(\mathcal{P})$. La liste d'isolation calculée va former l'ensemble des cellules de niveau 1.

Dans l'exemple qui nous occupe, l'échantillon de points pour les cellules de niveau 1 contient 5 éléments : les deux racines réelles du polynôme $X^2 - 1$ et un point par intervalle que ces racines déterminent (voir figure 3.16).

Faisons l'hypothèse que le procédé de dichotomie nous a permis de trouver la valeur exacte de la racine -1, et d'isoler le point 1 par l'intervalle $[\frac{1}{2}, \frac{3}{2}]$ [2].

L'échantillon triangulaire de niveau 1, étiqueté par les signes que prend le polynôme $X^2 - 1$, unique élément de $C_1(\mathcal{P})$, peut ainsi être :

$$(\{-2\}, +),\ (\{x_1 := -1\}, 0),\ (\{0\}, -),\ (x_2 \in [\ \frac{1}{2}, \frac{3}{2}\ ], 0),\ (\{2\}, +)$$

qui est représenté par la table des signes suivante :

	$-2 \in]-\infty, -1[$	x_1	$0 \in]-1, 1[$	x_2	$2 \in]1, \infty[$
$X^2 - 1$	$+$	0	$-$	0	$+$

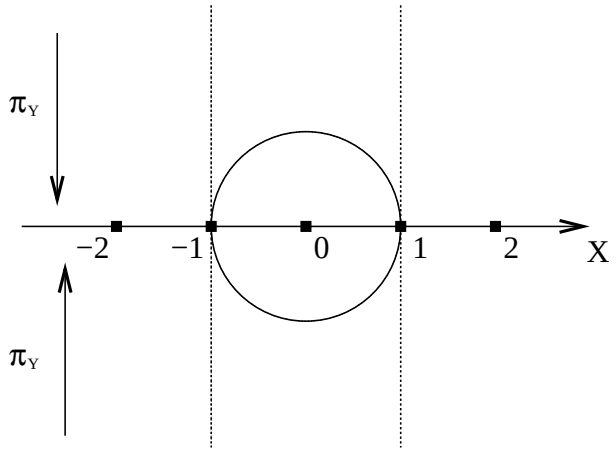


FIG. 3.16: *Cellules de niveau* 1 *pour le cercle unité*

Les points de cet échantillon sont représentatifs de l'ensemble des cellules de niveau 1. Ce sont des codages triangulaires (voir définition 3.2.2), de niveau un.

À présent, on doit effectuer une étape de remontée pour construire les cellule de niveau 2.

On va considérer successivement toutes les points de l'échantillon de niveau 1, et pour chacun d'entre eux, étudier l'unique polynôme de $C_2(\mathcal{P})$ sur la droite réelle de direction Y passant par le point considéré.

- Au dessus du point -2 : $P(-2, Y) = Y^2 + 3$ a un signe constant, positif.
  On peut vérifier qu'une droite réelle de direction Y, passant par un point quelconque de cette cellule de niveau 1 ne rencontre jamais le cercle. Le demi-plan $\{(x, y) \mid x < -1, y \in \mathbb{R}\}$ constitue donc une cellule de niveau 2.
- La situation est similaire au dessus du point 2 : le demi-plan $\{(x, y) \mid x > 1, y \in \mathbb{R}\}$ constitue une cellule.
- Au dessus du point $x_1 := -1$: $P(-1, Y) = Y^2$ a une unique racine (qui est en fait 0). Il y a donc trois cellules de niveau 2 au dessus de cette cellule de niveau 1 réduite à un singleton : deux demi-droites $\{(x, y) \mid x = -1, y < 0\}$ et $\{(x, y) \mid x = -1, y > 0\}$ et une cellule réduite à un singleton $\{\ (-1, 0)\ \}$.
  On peut vérifier que la droite réelle de direction Y, passant par le point -1 rencontre le cercle en un unique point.

[2] Cette hypothèse n'est pas très réaliste vue la symétrie du problème mais elle nous permettra d'illustrer les deux cas qui peuvent se présenter lors de l'isolation des racines

– La situation est similaire au dessus du point x_2, mais les calculs sont différents, car on ne connaît pas la valeur exacte de la racine, mais seulement sont encadrement par $\frac{1}{2}$ et $\frac{3}{2}$.
On doit en fait étudier $P(z, Y) = Y^2 + (z^2 + 1)$ où z est l'unique racine du polynôme $X^2 + 1$ dans l'intervalle $[\ \frac{1}{2}, \frac{3}{2}\ ]$.
Pour cela, on calcule les coefficients de Bernstein de $P(z, Y)$ dans les différentes bases de Bernstein qui interviennent et on compte à chaque fois les changements de signes dans la suite des coefficients.
Chacun de ces coefficients de Bernstein de $P(z, Y)$ est en fait un polynôme en X, évalués en le point z. Pour calculer le signe d' un tel coefficient, il faut savoir calculer le signe d'un polynôme en une variable en l'unique racine z du polynôme $X^2 - 1$ dans l'intervalle $[\ \frac{1}{2}, \frac{3}{2}\ ]$. Or on dispose d'un algorithme pour résoudre ce problème (voir figure 3.12).
– Au dessus du point 0 : $P(0, Y) = Y^2 - 1$. De manière analogue au calcul effectué au dessus du point -1, on trouve 5 cellules de niveau 2 : $\{(x, y) \mid y < \sqrt{1 - x^2}\}$, $\{(x, y) \mid y > \sqrt{1 - x^2}\}$, $\{(x, y) \mid y = \sqrt{1 - x^2}\}$, $\{(x, y) \mid y = -\sqrt{1 - x^2}\}$,

Ces 13 cellules sont représentées sur la figure 3.17.

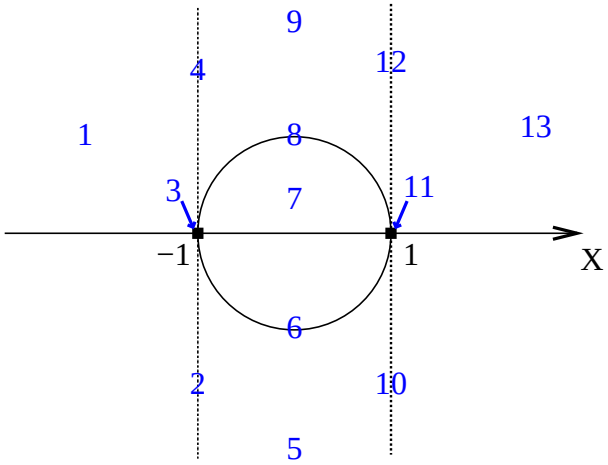


FIG. 3.17: *Cellules de niveau* 2 *pour le cercle unité*

Chapitre 4

Architecture d'une procédure de décision pour Coq

La mise en place de tous les ingrédients d'un algorithme de décomposition algébrique cylindrique pose plusieurs problèmes techniques. Le développement que nous avons réalisé est marqué par les spécificités du langage de programmation fonctionnelle avec types dépendants du système Coq, le souci d'efficacité et l'objectif de certification, qui modère les optimisations dans la première version de cette implémentation.

Dans ce chapitre, nous présentons trois aspects algorithmiques de la programmation du calcul de décomposition algébrique cylindrique : le traitement des coefficients rationnels, le choix des structures de données, et enfin le partage dans la structure d'échantillon cylindrique de points.

Nous décrivons également l'implémentation d'un prototype de tactique réflexive à partir de l'algorithme de CAD, ainsi qu'un apperçu des performances obtenues grâce à la réduction compilée vers une machine virtuelle (Grégoire et Leroy 2002; Grégoire 2003).

4.1 Abstraction de la représentation des coefficients

4.1.1 Développements sur l'arithmétique rationnelle en Coq

Il existe dans le système Coq plusieurs façons de calculer avec des coefficients rationnels.

Il existe dans la bibliothèque standard[1] une théorie des nombres rationnels, nommée QArith, et développée par P. Letouzey. L'origine de ce développement était de fournir une librairie d'arithmétique rationnelle pour l'extraction de la preuve du théorème de d'Alembert-Gauss formalisée dans le projet FTA (Cruz-Filipe et Letouzey 2006).

Dans cette implémentation, les nombres rationnels sont représentés comme des couples d'entiers en représentation binaire, c'est à dire par leur numérateur et leur dénominateur. Le numérateur est un entier relatif (du type Coq Z) et le dénominateur un entier strictement positif (du type Coq positive).

```
Record Q : Set := Qmake {Qnum : Z; Qden : positive}.
```

[1] depuis la distribution 8.1beta de juillet 2006. Auparavant, une version moins étoffée était disponible dans les contributions d'utilisateurs.

```
(* Le ``trait de fraction'' est noté par le symbole # *)
Notation "a # b" := (Qmake a b).
```

Les opérations proposées par défaut dans cette bibliothèque ne sont pas normalisantes : par exemple l'opération d'addition est programmée comme ceci :

```
Definition Qplus (x y : Q) :=
  (Qnum x * QDen y + Qnum y * QDen x) # (Qden x * Qden y).
```

Néanmoins, la bibliothèque fournit également une fonction de normalisation des fractions, qui met un rationnel sous forme de fraction irréductible. Dans le module Qcanon de la bibliothèque, se trouve ainsi une théorie des représentants canoniques des rationnels sous forme de fractions. Dans ce deuxième développement qui s'appuie sur la première version, les nombres rationnels sont représentés comme :

```
Record Qc : Set := Qcmake { this :> Q ; canon : Qred this = this }.
```

Le symbole :> signifie que l'on définit une cœrcion du type Qc vers Q.

Une approche complètement différente est adoptée dans les travaux de Y. Bertot et M. Niqui, qui proposent une arithmétique paresseuse basée sur la représentation de Stern-Brocot (Niqui et Bertot 2004). Dans cette contribution[2], les nombres rationnels sont représentés comme des mots finis sur un alphabet à deux lettres :

```
Inductive Q+ : Set := nR: Q+ → Q+ | dL: Q+ → Q+ | One: Q+.

Inductive Q : Set := Zero: Q | Qpos:  Q+ → Q | Qneg:  Q+ → Q.
```

Ces deux développements, et plus spécialement le dernier, sont bien adaptés pour la vérification formelle d'un algorithme qui sera exécuté après extraction vers un langage de programmation comme Ocaml ou Haskell. Par contre, l'exécution des calculs dans Coq sera peu efficace, voire totalement inadaptée dans le second cas, malgré les très bonnes performances des programmes extraits.

Plus récemment, une libraire d'arithmétique modulaire fonctionnelle certifiée a été développée en Coq (Grégoire et Théry 2006), qui permet de calculer efficacement sur les entiers en utilisant le mécanisme de réduction du système lui-même. Cette approche est plus adaptée à nos besoins.

L'idée principale de la représentation de B. Grégoire et L. Théry est la suivante : les algorithmes efficaces d'arithmétique entière (comme la multiplication de Karatsuba ou les divisions par blocs) reposent sur la possibilité de *découper* un nombre à faible coût, et exploitent une stratégie diviser pour régner. Par découpage des nombres, on entend ici l'isolation de blocs de chiffres dans la représentation dans la base choisie, comme par exemple lorsqu'on écrit

$$123456789 = 123 \times 10^6 + 456 \times 10^3 + 789$$

C'est pourquoi, plutôt que d'utiliser la représentation linéaire Z des entiers en base deux de la librairie standard de Coq, les auteurs introduisent une représentation sous forme

[2]Disponible à l'adresse http://coq.inria.fr/contribs/QArith-Stern-Brocot.tar.gz

d'arbres binaires, dont les feuilles sont en fait les chiffres de la représentation. Les feuilles sont ainsi étiquetées avec les éléments d'un type qui a autant de constructeurs que de chiffres dans la base.

À partir de cette arithmétique, on peut aisément dériver une implémentation des nombres rationnels, encore une fois représentés comme des couples formés d'un numérateur et d'un dénominateur, ce qui permet de travailler avec une implémentation des rationnels destinée au calcul efficace dans le système.

De fait, nous n'avons pas encore stabilisé notre choix pour la représentation des coefficients rationnels. Les tests effectués dans le système Coq sur notre algorithme de calcul de CAD semblent indiquer que la représentation de B. Grégoire et L. Théry, dans laquelle on normalise systématiquement les fractions, est de loin la plus efficace.

Néanmoins, il serait dommage de figer le développement sur un tel choix de coefficients, quand le fait de le rendre paramétrable offre aussi à l'utilisateur la possibilité de construire une tactique adaptée à son développement. On peut en effet imaginer d'autres domaines d'application que les implémentations citées ci-dessus comme par exemple une arithmétique flottante.

Nous avons donc décidé, dans le même esprit que lors de la conception de la tactique newring du chapitre 1, d'*abstraire* la structure de coefficients rationnels pour laisser ouverte cette capacité d'adaptation de l'outil.

4.1.2 Interface pour les coefficients rationnels

Concrètement, cette abstraction est mise en place grâce au système de modules de Coq (Chrzaszcz 2003).

On commence donc par écrire une signature pour la structure de rationnels, ce qui en Coq est un **Module Type**.

```
Module Type RAT_STRUCT.

  Parameter Rat : Set.

(* constantes, opérations sur les rationnels,
-MkRat construit un nombre rationnel a partir d'un entier et d'un
entier strictement positif
-RatEq est une relation d'egalite sur les rationnels
-rzero_test est un test a zero decidable
-rsign donne le signe d'un nombre rationnel :
 Z0 pour un rationnel nul, Zpos _, pour un positif, ...
-rpow construit une puissance d'un rationnel
*)

  Parameter  r0 : Rat.
  Parameter  r1 : Rat.
  Parameter  MkRat : Z → positive → Rat.
  Parameter  Norm : Rat → Rat.
  Parameter  radd : Rat → Rat → Rat.
  Parameter  ropp : Rat → Rat.
```

```
  Parameter  rprod : Rat → Rat → Rat.
  Parameter  rsub : Rat → Rat → Rat.
  Parameter  rinv : Rat → Rat.
  Parameter  rdiv : Rat → Rat → Rat.
  Parameter  rEq : Rat → Rat → Prop.
  Parameter  rzero_test : Rat → bool.
  Parameter  rlt : Rat → Rat → bool.
  Parameter  rsign : Rat → comparison.
  Parameter  rpow : Rat → N → Rat.
  Parameter  rabs_val : Rat → Rat.

End RAT_STRUCT.
```

Code 4.1: *Signature calculatoire pour la structure de coefficients*

Cette signature est redondante : on fournit par exemple un opérateur de multiplication `rprod` ainsi qu'un opérateur de puissance `rpow`. Ceci permet de ne pas imposer de contraintes inutiles à l'implémentation des opérations.

Par exemple, si l'on opte pour des opérations normalisantes sur les rationnels, en forme de fractions, l'opération de multiplication effectuera des tests et des calculs supplémentaires pour obtenir un résultat en forme de fraction irréductible. Par contre, si l'on fournit à l'opérateur de puissance un argument sous forme de fraction irréductible, l'élévation à la puissance de son numérateur et de son dénominateur donne toujours une fraction irréductible, d'où l'intérêt de dissocier ces deux opérations.

Pour pouvoir faire des preuves sur ces objets, il faut enrichir la signature avec les axiomes que l'on demande pour une telle structure de coefficients rationnels. Ces axiomes assurent en fait la correction du plongement `MkRat` de la structure de coefficients rationnels dans une représentation sous forme de fractions très similaire à celle de la bibliothèque standard `QArith`.

Ceci permettra ensuite de retrouver les propriétés désirées (structure d'anneau, intégrité,...) via des preuves sur les entiers.

On adopte dans la suite une série de notations (voir code 4.2).

```
  Notation "a ♯abs b" := (MkRat a b).

  Infix "++" := radd.
  Notation "-- x" := ropp.
  Infix  "**" := rprod.
  Infix "--" := rsub.
  Notation "// x" := (rinv x).
  Infix "//":= rdiv.

  Notation "x == y" := (rEq x y).
```

Code 4.2: *Notations pour la structure de rationnels*

Les axiomes de la structure se répartissent en plusieurs groupes.

Plongement dans la représentation fractionnaire

Tout ce qui suit repose sur la donnée d'une surjection `risMkRat` des représentations fractionnaires dans les coefficients rationnels de la structure abstraite.

```
Parameter risMkRat : ∀ x,
     { num_den :Z*positive | x == (fst num_den)♯abs(snd num_den)}.
```

Égalité, test à zéro :

La relation d'égalité `rEq` doit être une relation d'équivalence. Ainsi la structure de rationnels sera munie d'une égalité de setoïde.

```
Parameter req_refl : ∀ x, x==x.
Parameter req_sym : ∀ x y, x == y ⇒ y == x.
Parameter req_trans : ∀ x y z, x == y → y == z ⇒ x == z.
```

Enfin, on donne la sémantique de l'égalité et du test à zéro sur les fractions :

```
Parameter req_def : ∀ x y x' y',
  x♯abs y == x'♯abs y' ⇔ x*(Zpos y') = (Zpos y) *x'.

Parameter rzero_test_def : ∀ x y,
  rzero_test (x♯absy) = match x with Z0 → true |_ → false end.
```

Et la compatibilité de l'opération de test à zéro avec la relation d'égalité.

```
Parameter r0_test_ext : ∀ x1 x2,
  x1 == x2 → rzero_test x1 = rzero_test x2.
```

Relation d'ordre :

De la même façon, on donne la sémantique de l'opérateur de comparaison :

```
Parameter rlt_def : ∀ x y x' y',
 rlt (x♯abs y)  (x'♯absy')
             =
 match  (((Zpos y)*x') ?= x*(Zpos y'))  with
   | Gt → true |_ → false end.
```

et sa compatibilité avec la relation d'égalité.

```
Parameter rlt_ext : ∀ x1 x2 y1 y2,
 x1 == x2 ⇒ y1 == y2 ⇒ rlt x1 y1 = rlt x2 y2.
```

De même pour l'opération de signe :

```
Parameter rsign_def : ∀ x y,
 rsign (x♯abs y) = match x with
                     |Z0 → Eq
                     |Zpos _ → Gt
                     |Zneg _ → Lt
                     end.

Parameter rsign_ext :  ∀ x1 x2, x1 == x2 ⇒ rsign x1 = rsign x2.
```

Constantes et opérations :

On procède de même avec les opérations de corps de la structure :

```
Parameter r1_def : r1 == 1♯abs 1 .

Parameter r0_def : r0 == 0♯abs1.

Parameter radd_def : ∀ a b c d,
  (a♯abs b) ++ (c♯abs d) == (a*(Zpos d) + c*(Zpos b))♯abs(b*d).

Parameter rsub_def : ∀ x y, x -- y == x ++ (-- y).

Parameter rop_def : ∀ a b, -- (a♯abs b) == (- a)♯abs b.

Parameter rinv_def_pos : ∀  a b ,
  rinv ((Zpos a)♯abs b) == (Zpos b)♯abs a.

Parameter rinv_def_neg : ∀ a b,
  rinv ((Zneg a)♯abs b) == (Zneg b)♯abs a.

Parameter rprod_def : ∀ a b c d,
  (a♯abs b) ** (c♯abs d) == ((a * c)♯abs(b*d).
```

4.1.3 Propriétés de la structure des coefficients rationnels

Une fois déclarée la signature de la structure de coefficients rationnels, on peut déjà dériver une série de conséquences des axiomes qui seront communes à toutes les implémentations des coefficients rationnels partageant cette signature. On va donc définir un premier *foncteur*, c'est à dire une application qui à tout module de signature RAT_STRUCT associe un module contenant les énoncés et les preuves de ces propriétés.

On déclare un foncteur RAT_PROPS :

```
Module RAT_PROPS(Q:RAT_STRUCT).
Import Q.
```

Après avoir déclaré la structure de setoïde et les différents morphismes rendu disponibles par les axiomes de compatibilité, on peut maintenant prouver :

- Un lemme trivial mais utile de destruction pour les rationnels en forme fractionnaire :

```
Lemma rdestruct : ∀ a b c d, a = c ⇒ b = d ⇒ (a♯abs b) == (c♯abs d).
```

- Les axiomes de la structure d'anneau : par exemple,

```
Lemma radd_0_l    : ∀ x, r0 ++ x == x.
   Proof.
    intro x.
    elim (risMkRat x);intros u x_def.
    rewrite r0_def.
    rewrite x_def.
    rewrite radd_def.
```

```
    simpl;apply rdestruct;[ring|reflexivity].
  Qed.
```

La preuve des propriétés d'anneau permet de pouvoir déclarer une *structure abstraite d'anneau* : qui instancie la tactique générique décrite dans la section 1.2 du chapitre 1 pour donner un outil d'automatisation des preuves d'égalité dans la structure de coefficients rationnels, que l'on appelle à l'aide de la commande `setoid ring`.

– L'intégrité de la structure de rationnels, que l'on obtient grâce à celle des entiers relatif `Z` de Coq :

```
Lemma rintegral : ∀ x,
  ¬ x==r0 ⇒ ∀ y, ( x ** y == r0 ⇒ y == r0).
```

– et bien d'autres résultats élémentaires : transitivité de l'ordre `rlt`, régularité des opérations,...

4.1.4 Formalisations du corps des réels

Le choix des spécifications des opérateurs que comporte la signature de la structure de coefficients n'est pas encore stabilisé à ce stade de notre travail. En particulier, il est nécessaire pour les preuves de plonger les coefficients dans une théorie des nombres réels. Cette théorie des nombres réels est le domaine d'application de la tactique.

Les développements visant à formaliser le corps des nombres réels dans un assistant à la preuve peuvent être classifiés selon deux critères indépendants : ils sont sont d'une part soit *constructifs*, soit *classiques*, et d'autre part ils reposent soit sur une *axiomatique*, soit sur une *construction* des nombres réels.

	Systèmes	Classique	Constructif
Axiomatique	Coq	• Reals (Mayero 2001)	• FTA (Geuvers et Niqui 2002)
	Autres systèmes	PVS, Isabelle	
Construction	Coq		• FTA (Geuvers et Niqui 2002) • Réels exacts coinductifs (Bertot 2006) (Ciaffaglione et Gianantonio 2000)
	Autres systèmes	HOL, HOL-Light, Mizar, Nuprl, Lego	Nuprl

TAB. 4.1: *Quelques formalisations de l'arithmétique réelle dans les systèmes de preuve formelle*

Le choix d'un développement classique est conforme à la présentation la plus fréquente dans la littérature mathématique. Dans ce contexte, le corps des nombres réels est un corps commutatif, totalement ordonné, archimédien et complet. On dispose alors en fait d'une structure de corps réels clos discret archimédien. Ces propriétés peuvent être posées

comme axiomes définissant un nouvel objet pour le système. C'est le choix qui est fait dans la bibliothèque standard du système Coq (Mayero 2001), dans la théorie développée dans le système Lego (Jones 1991) ainsi que pour les nombres réels du système PVS.

Il est également possible de *construire* cet ensemble de nouveaux objets, par exemple à partir d'une formalisation préexistante des nombres rationnels. Il existe plusieurs façons de réaliser une telle construction. Les deux plus utilisées sont certainement la méthode de Cantor, qui construit les nombres réels à partir de suites de Cauchy de nombre rationnels, et la méthode des coupures de Dedekind, qui définit un nombre réel par l'ensemble des rationnels qui lui sont strictement inférieurs. Dans ce contexte classique, il n'existe pas de distinction entre les objets calculables et ceux qui ne le sont pas (signe d'un nombre réel, borne supérieure).

Dans les assistants à la preuve HOL et HOL-Light, la construction d'une nouvelle structure (plutôt que son axiomatisation) garantit que l'extension du système obtenue ne brise pas sa cohérence. Dans ces deux systèmes, les nombres réels sont construits classiquement à l'aide des coupures de Dedekind (Harrison 1994).

Le point de vue constructif quant à lui s'astreint à ne définir que des objets, et en particulier des opérations, pour lesquels on sait écrire un algorithme qui en calcule la valeur. Ainsi, dans le cas général, on ne sait pas construire une preuve finie de l'égalité de deux nombres réels. La propriété d'ordonnabilité du corps des nombres réels tombe alors en défaut, et la relation d'égalité est souvent remplacée par celle de distinguabilité $\perp$ définie comme :

- $\forall x, \neg x \perp x$
- $\forall xy, x \perp y \Rightarrow y \perp x$
- $\forall xy, x \perp y \Rightarrow \forall z, x \perp z \vee y \perp z$

Cette relation coïncide avec la négation de l'égalité dans la théorie classique.

Un développement constructif a du sens dans un assistant à la preuve basé sur une logique intutionniste. Ainsi, on trouve une telle construction dans le système Nuprl (Howe 1986), ainsi que dans le système Coq (Geuvers et Niqui 2002). Ces derniers travaux comprennent à la fois une axiomatique intutionniste (tous les axiomes ont un contenu calculatoire) et la construction d'un modèle par les suites de Cauchy. Il existe un autre développement en Coq qui propose une axiomatique intuitionniste puis un modèle (Ciaffaglione et Gianantonio 2000). Ce développement utilise des types coinductifs et des flots de chiffres ternaires pour construire le modèle des réels exacts. Enfin, des travaux plus récents (Bertot 2006) utilisent également une représentation coindutive par des flots de chiffres pris dans un alphabet fini ; les propriétés de ces nombres réels sont montrés par injection dans la bibliothèque standard de Coq, qui est prise comme référence.

La tactique que nous avons décrite permet de décider la théorie classique de l'arithmétique réelle du premier ordre. D'autre part la théorie intuitionniste des corps réels clos est indécidable (Gabbay 1973). Quel est alors l'apport amené par un tel outil d'automatisation pour les formalisations constructives ? La diversité de ces développements, tant dans leur conception que dans leurs finalités plaide au contraire pour le conception d'un outil d'automatisation modulable, qui permette d'adapter la théorie des nombres réels de la même façon qu'il permet un choix dans l'implémentation des coefficients. Il serait en effet dommage de figer la tactique en la spécialisant sur un modèle ou même en la rendant inexploitable dans un cadre intuitionniste. Dans un développement constructif, la tactique perd bien sûr sa complétude, pour une raison intrinsèque à la théorie sur

laquelle elle opère, mais elle ne perd pas tout intérêt.

Pour pouvoir abstraire l'implémentation du modèle des réels utilisé, il faut préciser l'axiomatique des réels utilisée par la preuve de correction de la procédure de décision. La question est en fait de savoir quelle classe de $\mathcal{P}$-formules peuvent être décidées à partir d'un échantillon cylindrique calculé pour la famille $\mathcal{P}$.

Tout d'abord, la correction du calcul d'un échantillon cylindrique de points exhaustif, et des signes réalisés par la famille en ces points se fait sans utiliser d'arguments classiques. Pour toute formule existentielle du premier ordre qui est vraie dans la théorie (intuitionniste) des réels, l'algorithme construit, effectivement un témoin pour la formule. Par contre, pour prouver une formule quantifiée universellement à partir de l'échantillon cylindrique de points, on doit revenir à un corps réel clos discret, c'est à dire utiliser l'axiome de trichotomie, qui exprime que l'ordre est total sur l'ensemble des nombres réels. En effet cet argument est indispensable pour exprimer que la partition calculée est exhaustive. L'énoncé même de l'axiome est en fait une $\{X\}$-formule universelle, qu'on ne peut décider constructivement.

Pour résumer, le caractère classique de la preuve de correction de cette procédure de décision :

- est due à l'utilisation de l'axiome de trichotomie
- n'est nécessaire qu'au moment de l'interprétation de l'échantillon de points, pour traduire son caractère exhaustif
- n'interviendra pas pour les preuves du fragment existentiel.

On peut exprimer différemment le fait que seule l'exhaustivité de l'échantillon nécessite un argument classique en caractérisant les formules validée par la procédure de décision comme celles qui n'ont pas de contre-exemple classique.

Dans le Calcul des Constructions Inductives, qui est la théorie des types implémentée par le système Coq, les citoyens de première classe se voient attribuer l'une des deux sortes **Set** ou **Prop**. Le sens informel de cette distinction est le suivant : la sorte **Set** contient le *contenu calculatoire* d'une formalisation, qui dit pouvoir être extrait vers un langage de programmation afin d'être exécuté. Cette extraction se fait en *effaçant* du développement la partie qui contient l'*information logique*, qui doit être placée dans la sorte **Prop**.

Néanmoins, cette distinction va au delà d'une simple convention, et les règles de typage qui gouvernent la bonne formation des termes sont différentes pour ces deux sortes. L'une des caractéristiques de la sorte **Prop**, réside dans le fait que les règles de typage du calcul ne permettent pas de construire un objet résidant dans la sorte **Set** par filtrage sur un autre objet qui habiterait dans la sorte **Prop**. On dit que la sorte **Prop** est munie d'un schéma d'élimination faible. Toutefois, le système de typage de Coq *ne garantit pas la calculabilité*. De fait, dans les développements axiomatiques de la table 4.1 réalisés en Coq, on observe des choix différents pour la sorte de l'axiome de trichotomie. Dans la bibliothèque standard, l'axiome est énoncé dans une version forte :

```
Axiom total_order_T : ∀ r1 r2:R, {r1 < r2} + {r1 = r2} + {r1 > r2}.
```

La syntaxe ci-dessus signifie que cet axiome est placé dans la sorte **Set**, autrement dit, on s'autorise à définir des *fonctions* par filtrage sur le signe d'un réel, en particulier un test à zéro booléen.

Dans l'axiomatique constructive (Geuvers et Niqui 2002), il n'y a pas de tel axiome

dans la sorte **Set**, ni même en fait dans la sorte **Prop**. En notant [#] la relation de distinguabilité définie ci-dessus, on remplace la trichotomie par le fait que l'on sait comparer deux réels si l'on a prouvé qu'ils étaient distincts.

```
ax_less_conf_ap : ∀ x y, ⇔ (x [#] y) (less x y or less y x)
```

On ne brise pas la cohérence du système en choisissant une sorte ou l'autre pour l'axiome de trichotomie.

Par contre, l'avantage d'une axiomatique de corps réel clos discret dans laquelle l'axiome `total_order_T` serait placé dans la sorte **Prop** serait le suivant : une telle structure n'autorise de définir dans la sorte **Set** que les objets pour lesquels on dispose d'un moyen de construction *effectif*. Pour la description des *faits*, on utilise la sorte **Prop**, et le système de type empêche d'utiliser un *fait* pour décrire une méthode effective. Les objets définissables dans la sorte **Set** sont les mêmes que ceux que l'on peut définir dans une structure de corps réel clos intuitionniste.

Notre point de vue actuel sur l'axiomatisation de la structure abstraite sur laquelle la tactique travaille est le suivant : le premier outil à construire doit reposer sur une axiomatisation de corps réel clos discret archimédien dont l'axiome de trichotomie est placé dans la sorte **Prop**. Dans un deuxième temps, il est possible d'adapter l'outil pour une structure de corps réel clos intuitionniste, i.e. dans laquelle on a supprimé l'axiome de trichotomie, qui, avec le même programme de décision et la même preuve de correction pourra être appliquée au fragment existentiel de la théorie du premier ordre.

4.2 Structure du développement, structures de données

L'algorithme de décomposition algébrique cylindrique repose sur une récurrence sur le nombre de variables des polynômes dans les atomes de la formule à décider. Le cas de base est celui des polynômes à une seule variable et à coefficients rationnels.

Pour un problème à $n+1$ variables, on doit calculer en fonction de n :

- des types de données, principalement :
  - celui des polynômes ;
  - celui des codages triangulaires (voir définition 3.2.2 du chapitre 3) ;
  - celui de l'échantillon cylindrique calculé.
- des opérations sur ces types de données :
  - opérations d'anneaux sur les polynômes ;
  - ...
  - calcul d'un échantillon cylindrique exhaustif.

Tous ces objets sont dépendants de n et construits par récurrence, le cas de base des polynômes à une variable utilisant la structure de coefficients rationnels qui paramètre toute la procédure.

Malheureusement, le système de modules de Coq n'est pas adapté à ce type de problèmes, du fait de la construction par récurrence. On utilise donc une structure d'enregistrement dépendant pour mettre en place cette construction.

La première solution consiste à construire successivement des enregistrements qui prennent en paramètre un entier n représentant la dimension du problème. Dans ce cas,

l'enregistrement construit possède des champs dans la sorte **Set** (pour les types de données) et des champs qui représentent les différentes opérations nécessaires à la récursion :

```
Record Cad1(n:nat) : Set := mk_cad1{
  Coef:Set;
  ...
  Pol_add : Pol → Pol → Pol; ...}
```

Le choix de cette première solution s'est avéré inadapté. Avec la version du système Coq utilisée, le point fixe final qui devait construire l'enregistrement pour toute valeur du paramètre n n'était pas accepté par le système en un temps raisonnable : le calcul de la bonne formation de ce point fixe prenait un temps et des ressources CPU considérables et de fait nous ne l'avons pas vu terminer.

Pour contourner ce problème, nous avons adopté une démarche différente et finalement plus proche de la construction mathématique standard d'une telle récursion. Il s'agit en fait de remarquer que puisque la récurrence sur laquelle repose la construction n'est pas forte, on peut en fait se passer de la dépendance en n et ne travailler en fait qu'avec les deux niveaux jamais considérés dans la preuve : l'hypothèse de récurrence et le niveau à construire.

En pratique on *sépare* le calcul des types de données de celui des opérations. Sous l'hypothèse qu'on dispose des types de données construits pour le niveau n, on programme des constructeurs de types qui construisent les types de données correspondant pour le niveau n+1. Ces types de données du niveau n sont également donnés en paramètres à l'enregistrement qui rassemble les opérations programmées pour le niveau n+1.

Finalement, pour chaque type de données ainsi que pour l'enregistrement, la version dépendante du paramètre n est construite par un point fixe qui itère le constructeur de types.

Tout d'abord nous donnons dans la section 4.2.1 la méthode à deux niveaux suivie pour construire les types de données dépendants. Ensuite, dans le section 4.2.2, nous exposons le principe de la construction des opérations sur ces types de données.

4.2.1 Types de données

La première étape est de définir les types de données sur lesquels les fonctions à programmer vont opérer. Dans cette section, on explique comment sont programmés les constructeurs de types, qui prennent en paramètre les types de données du niveau n pour fabriquer les types du niveau n+1. Puis on donne les points fixes qui itèrent cette construction dans la phase finale d'assemblage du code, qui se trouvent dans un module compilé distinctement.

Polynômes

Le première exemple illustrant cette construction est celui du type des polynômes. En fait on a déjà donné le constructeur de types correspondant au chapitre 1, dans le code 1.1 :

```
(* Type des coefficients *)
```

```
Variable Coef : Set.

Inductive Pol1:Set:=
  |Pc : Coef → Pol1 Coef
  |PX : Pol1 Coef → Coef → Pol1 Coef.
```

Dans tout le développement des opérations de niveau $n+1$, on n'aura accès qu'au type `Coef`, (qui est en fait le type des polynômes de niveau n) et au type `Pol1 Coef`.

Naturellement, pour $n = 0$, le type construit doit être celui des coefficients rationnels, c'est à dire des coefficients de base, qui est le champ `Rat` du module de signature `RAT_STRUCT` donné en paramètre.

On peut maintenant programmer le point fixe qui fabrique les polynômes à n variables pour une valeur arbitraire de n.

```
Fixpoint Poln(n:nat):Set:=
  match n with
    |O ⇒ Rat
    |S n ⇒ Pol1 (Poln n)
  end.
```

Sur le même modèle, on construit un type qui retient les informations sur un polynôme qui sont calculée souvent dans les différentes étapes du calcul de CAD. En fait pour chaque nouveau polynôme calculé, que ce soit dans un calcul d'élimination de variables, ou dans un calcul de coefficients de Bernstein, on mémorise un quintuplet contenant :

- La valeur du polynôme ;
- Sa partie sans carrés ;
- Le degré de sa partie sans carré ;
- Le signe du polynôme en $-\infty$;
- Le signe du polynôme si l'on sait qu'il est constant.

Pour chaque niveau n, ce type `Info` est construit de façon identique à la construction des polynômes. Pour $n = 0$, il est égal au type des coefficients rationnels, qui ne nécessitent pas un tel traitement.

Codages triangulaires algébriques

Pour calculer la CAD d'une famille de polynômes $\mathcal{P}$ à $n+1$ variables, l'algorithme manipule des points de $\mathbb{R}^n$ qui sont issus d'un échantillon exhaustif de points calculé pour la famille $Elim(\mathcal{P})$. Ces points de $\mathbb{R}^n$ ont des coordonnées algébriques (qui peuvent être rationnelles) et on les représente sous la forme de codages triangulaires.

Les codages triangulaires de dimension $n+1$ sont construits à partir des codages de dimension n, en ajoutant l'information qui définit la dernière coordonnée.

Au niveau $n+1$, on dispose du type `coefinfo`, qui la valeur de `Info` pour le niveau n et de `Coef` le type des coefficients. Le cas algébrique de la $n+1$ème coordonnée des codages triangulaires de dimension $n+1$ est défini en Coq de la façon suivante :

```
(* Type des coefficients *)
Variable Coef : Set.
(*Type des informations retenues pour les coefficients *)
Variable coefinfo : Set.
```

```
Let Pol := Pol1 Coef.
...
Definition mkAlg := uple5 Rat Rat Pol Pol (list coefinfo).
```

Le constructeur `uple5` est un constructeur de quintuplets polymorphes. Un élément de type `mkAlg` contient les bornes rationnelles a et b de l'intervalle, le polynôme annulateur P, sa partie sans carrés $\overline{P}$ et les coefficients de Bernstein de $\overline{P}$ sur $]a, b[$.

On passe ensuite simplement d'un codage triangulaire `path` de dimension n à un codage triangulaire `next_path` de dimension $n+1$ en ajoutant une coordonnée :

```
(* Type des coefficients *)
Variable Coef : Set.
(*Type des informations retenues pour les coefficients *)
Variable coefinfo : Set.
(* Type des codages triangulaires en dimension n1 *)
Variable path : Set.
...
(* mkRpoint est soit un mkAlg, soit un nombre rationnel *)
Let next_label := mkRpoint.

Definition next_path := (path * next_label).
```

Le point fixe qui itère cette définition utilise les types de données des polynômes et des informations retenues pour les polynômes.

```
Fixpoint pathn(n:nat):Set:=
  match n with
    |O⇒unit
    |S n ⇒ next_path Rat (Poln n) (Infon n) (pathn n)
  end.
```

Échantillon cylindrique

L'algorithme de décomposition algébrique cylindrique calcule, pour une famille de polynômes à n variables, un échantillon cylindrique de points de $\mathbb{R}^n$ qui est représenté de façon arborescente (voir la définition 3.1.9 du chapitre 3). On rappelle que les nœuds internes de ces arbres sont étiquetés par des nombres réels, et leur feuilles, par des listes de signes. En fait on étiquette les feuilles par une liste de couples formés d'un polynôme et de son signe.

Pour une famille $\mathcal{P}$ de polynômes à $n+1$ variables, on calcule récursivement un arbre de profondeur $n+1$ qui correspond à la famille de polynômes à n variables $Elim(\mathcal{P})$. À partir de celui-ci, on calcule l'arbre de profondeur $n+2$ qui décrit l'échantillon cylindrique pour la famille $\mathcal{P}$.

Sur la figure 4.1 on a représenté un échantillon cylindrique de point en dimension n. Les noeuds internes étiquetés par des nombres réels sont représentés par des carrés. Les feuilles, qui sont étiquetées par des listes de couples polynôme et signe, sont représentées par des cercles.

Pour chaque feuille de l'échantillon cylindrique, on peut associer au chemin de la racine vers cette feuille un point $\overline{z} = (z_1, \ldots, z_n)$ de $\mathbb{R}^n$. Les coordonnées de ce point $\overline{z}$ sont les

étiquettes portées par les noeuds du chemin : la première coordonnée est l'étiquette du noeud de profondeur 1, etc.

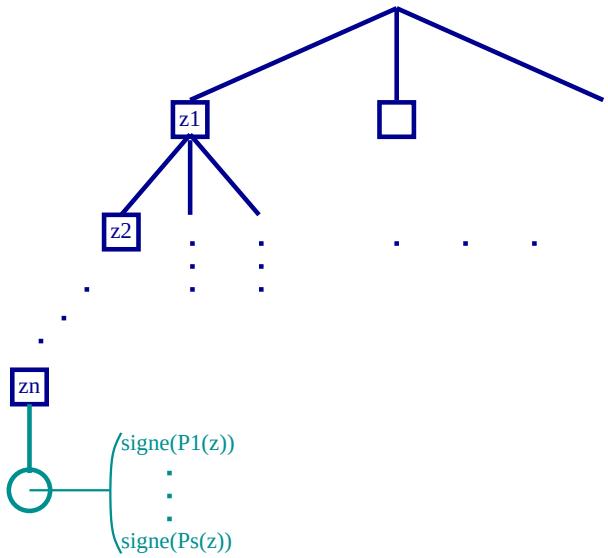


FIG. 4.1: *Échantillon cylindrique de points en dimension n*

Pour calculer l'échantillon cylindrique de points de la famille $\mathcal{P}$, pour chaque $\overline{z}$ associé à un chemin, l'algorithme calcule une subdivision de la droite $\overline{z} \times \mathbb{R}$ et une liste de points $\{x_1, \ldots, x_n\}$ contenant un point par intervalle de cette subdivision.

Ainsi dans l'ensemble des points de l'échantillon, chaque élément $\overline{z} \in \mathbb{R}^n$ est remplacé par la liste des points $(\overline{z}, x_1), \ldots, (\overline{z}, x_n)$.

Dans la structure arborescente, cela signifie que chacune des feuilles portant les signes des polynômes de $Elim(\mathcal{P})$ est remplacé dans le nouvel arbre par un "bourgeon" de profondeur 1, c'est à dire une liste de noeuds internes, qui sont chacun père d'une feuille portant les signes des polynômes de la famille $\mathcal{P}$. Ce processus est illustré sur la figure 4.2.

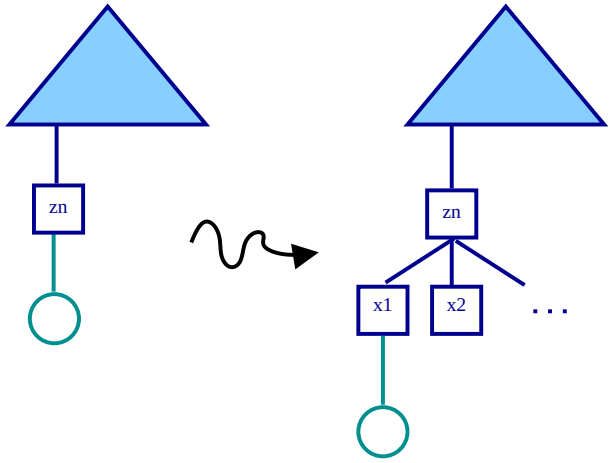


FIG. 4.2: *Construction d'un échantillon de dimension n + 1*

Le type des échantillons de points de niveau n doit lui aussi être construit par récursion

comme un type dépendant de n. En effet un noeud de profondeur k est étiqueté par la $k$ème coordonnée d'un codage triangulaire. Le type de cette étiquette dépend ainsi de k, c'est en fait le type `mkRpoint` ci-dessus.

On utilise encore une fois le principe des deux niveaux pour construire ces arbres polymorphes. Remarquons que ces arbres poussent par les feuilles, et non pas par la racine. Par conséquent, on ne construit pas directement le type des échantillons de points de niveau n, mais un type des échantillons de points de niveau n à paramètre.

Pour obtenir le type des échantillons de points de niveau n, on instancie le paramètre par le type des listes de couples `list (mkRpoint * (Pol * Sign))`. Pour obtenir le type des échantillons de points de niveau supérieur, on instancie le paramètre par le type des "bourgeons".

```
(* Hypothese de récurrence au niveau n :
le type donne en argument est celui des feuille de l'arbre *)
Variable mkCad:Set → Set.
```

Si l'on dispose d'un constructeur de type qui fabrique le type des échantillons de points de niveau n étant donné le type `Leaf_type` de ses feuilles, alors on sait donner le constructeur analogue pour le niveau n (voir code 4.3).

```
(* Le constructeur du type des échantillons de profondeur n,
qui prend en argument le type des feuilles de l'arbre *)
Variable mkCad:Set → Set.

(* le type des nouvelles coordonnees algebriques. C'est le type des
noeuds de profondeur n dans un echantillon de niveau n+1 *)
Let next_label := mkRpoint.

(* Le constructeur du type des echantillons de profondeur n+1, dont
les feuilles ont le type Leaf_type*)
Definition next_mkCad(Leaf_type:Set):=
                              mkCad (list (next_label * Leaf_type)).
```

Code 4.3: *Construction du type des échantillons de points*

Un échantillon cylindrique de niveau 1 est constitué d'une liste de points algébriques, et ses signes réalisés en chacun de ces points par les polynômes de la famille. L'opérateur `mkCad` est initialisé avec l'identité :

```
Fixpoint mkCadn(n:nat):Set → Set:=
  match n with
    |O ⇒ fun A :Set ⇒ A
    |S n ⇒ next_mkCad Rat (Poln n) (Infon n) (mkCadn n)
  end.
```

Il s'agit maintenant de décrire la construction des opérations manipulant ces types de données, et la mise en place complète du procédé de récursion.

4.2.2 Opérations

Lorsqu'on dispose des types de données du niveau `n`, par hypothèse de récurrence, et qu'on a construit les types de donnés du niveau `n+1`, on dispose de toute l'information nécessaire pour écrire le type des fonctions que l'on doit programmer au niveau `n+1`.

On peut donc définir le type de l'*enregistrement* qui sera construit pour chaque valeur de `n` et qui rassemble les fonctions construites par récurrence sur n. En particulier l'un de ses champs est la fonction qui construit un échantillon cylindrique exhaustif pour une famille de polynômes.

Le type de cet enregistrement, représenté sur le code 4.4, ne dépend pas lui non plus de `n`. Il prend en paramètre les types du niveau `n` et utilise les types du niveau `n+1` préalablement construits. Contrairement aux exemple de la section 4.2.1, le type de l'enregistrement pour un niveau `n+1` donné ne prend pas comme paramètre le type de l'enregistrement construit au niveau `n`.

Une description détaillée des opérations ci-dessus prendrait beaucoup de place, nous renvoyons le lecteur intéressé aux commentaires insérés dans le code `Coq`.

Remarquons simplement que certaines des opérations, comme `path_refine` qui raffine un encodage triangulaire, prennent en argument un entier. Cet entier représente une borne sur le nombre de pas de dichotomie autorisé dans les calculs d'isolation. Les fonctions qui prennent un tel argument renvoient ainsi un type option qui reflète l'échec possible du calcul par manque de ressources. On peut supprimer cet aspect peu satisfaisant du code en prévoyant le nombre de pas de dichotomie nécessaire à partir de la famille de départ. C'est en effet possible puisque la borne recherchée est en fait fonction de la distance minimale entre deux racines distinctes des polynômes de la famille, et que cette quantité s'estime en fonction des coefficients des polynômes. Nous avons eu recours à cette solution provisoire pour des considérations de simplicité, mais nous prévoyons de supprimer ces options incessamment, simplement en programmant le calcul de la borne.

Pour un niveau n donné, l'ensemble des opérations, qu'elles soient des champs de l'enregistrement ou bien des opérations "auxiliaires", se répartit en trois catégories :

- Celles qui sont spécifiques au cas de base des polynômes à une variable sur des coefficients rationnels.
- Celles qui sont spécifiques au cas des polynômes à plusieurs variables.
- Celles que l'on peut construire à partir d'hypothèses sur l'ensemble des coefficients qui seront réalisées à la fois dans le cas de coefficients constants rationnels et dans le cas de coefficients polynomiaux.

Par exemple, la programmation des opérations d'anneau pour les polynômes à une variable sur une structure d'anneau de nécessite seulement la donnée des opérations d'anneau sur les coefficients, et cette construction ne dépend pas de la nature des coefficients considérés.

Par contre les opérations d'élimination d'une variable n'ont de sens que pour les dimensions supérieures.

Malheureusement on ne peut pas mettre en place le partage de ce code dans la configuratin actuelle du système de modules de `Coq`. Un mécanisme de propagation des paramètres de modules définis dans des sections pourrait par exemple permettre l'abstraction du code commun au cas de base et au cas récursif par un foncteur, mais cette possibilité n'est pas disponible à ce jour.

```
(* Type des coefficients rationnels *)
Variable Rat : Set.
(* Type des coefficients des polynomes *)
Variable Coef : Set.
Let Pol := Pol1 Coef.
...
(* Type des informations necessaire au calcul de signe *)
Let build_Info := uple5 Pol Pol N Sign Sign.
...
(* Type des chemins dans l'arbre des echantillons de niveau n *)
Variable path : Set.
Let next_label := mkRpoint.
Definition next_path := (path * next_label)

(* Constructeur du type des  echantillons de niveau n, etant donne le
type des feuilles *)
Variable mkCad:Set → Set.

Definition next_mkCad(C:Set):= mkCad (list (next_label * C)).

 Record Cad : Set := mk_cad {
    Pol_0                : Pol;
    Pol_1                : Pol;
    Pol_add              : Pol → Pol → Pol;
    Pol_mul_Rat          : Pol → Rat → Pol;
    Pol_mul              : Pol → Pol → Pol;
    Pol_sub              : Pol → Pol → Pol;
    Pol_opp              : Pol → Pol;
    Pol_of_Rat           : Rat → Pol;
    Pol_is_Rat           : Pol → bool;
    Pol_zero_test        : Pol → bool;
    Pol_base_cst_sign    : Pol → Sign;
    Pol_pow              : Pol → N → Pol;
    Pol_div_Rat          : Pol → Rat → Pol;
    Pol_div              : Pol → Pol → Pol;
    Pol_gcd_gcd_free     : Pol → Pol → Pol * Pol * Pol;
    Pol_square_free      : Pol → Pol;
    path_refine          : next_path → nat → option next_path;
    Pol_low_bound        : Pol → path → Rat;
    Pol_value_bound      : next_path → Pol → Rat * Rat;
    Info_of_Pol          : Sign → Pol → build_Info;
    Pol_low_sign         :  path → Pol → Pol → nat →
      (path * (Sign));
    Pol_sign_at          : build_Info → next_path → nat →
      next_path * (Pol * Sign)%type;
    Pol_cad : list Pol → nat → next_mkCad  (list (Pol * Sign))  }.
```

Code 4.4: *Enregistrement calculé par la procédure*

La partie purement calculatoire du développement s'organise en cinq modules *compilés* (dont le contenu est rendu disponible par l'instruction `Require`) et un bloc de quatre fichiers *chargés* (dont le contenu est rendu disponible par l'instruction `Load`). Cette distinction permet de factoriser le code partagé par le cas de base et le cas récursif de la procédure.

La répartition est la suivante :

- Le module `CAD_types` contient les définitions des constructeurs de types décrits dans la section 4.2.1.
- Le module `Tarski` contient les définitions des points fixes qui itèrent les constructions de types de données, jusqu'au point fixe qui construit pour chaque valeur de n un enregistrement `CAD`.
- Le module `One_dim` contient un foncteur qui prend en argument une structure de coefficients rationnels. Le module obtenu définit le cas de base de la récurrence, c'est à dire un enregistrement nommé `One_dim_cad`, de type `Cad` pour le problème à une variable.
- Le module `Up_dim` contient un foncteur qui prend en argument une structure de coefficients rationnels. Il définit le constructeur de type pour fabriquer un enregistrement de type `Cad` contenant les opérateurs du problème à $n+1$ variables sous l'hypothèse qu'on dispose d'un enregistrement de type `Cad` contenant les opérateurs du problème à n variables.
- Les fichiers `Pol_on_Coef`, `Pol_on_Coef_on_Rat`, `Bern` et `Alg` contiennent du code partagé par les deux modules `One_dim` et `Up_dim`.
  Encore une fois, la dépendance récursive des opérations, qui impose une construction par point fixe sur un entier rend inutilisable le système de modules dans ce contexte. Le partage du code se fait grâce à des conventions de nommage dans les variables : on a pris soin dans le préambule des modules `One_dim` et `Up_dim` respectivement, de donner le même nom aux constantes utilisées par le code partagé contenu dans ces fichiers. Par exemple, le coefficient constant nul est nommé `c0` dans les deux modules `One_dim` et `Up_dim`, alors qu'il est instancié par le rationnel nul dans le premier cas et par le champs `P0` de la variable de type `Cad` dans le second cas.
  La séparation en quatre fichiers a été faite pour faciliter la hiérarchie des preuves : on a rassemblé
  - Dans `Pol_on_Coef`, les opérations construites pour les polynômes en formes de Horner creuse à partir des opérations disponibles sur un anneau de coefficient : égalité, opérations d'anneau,... ;
  - Dans `Pol_on_Coef_on_Rat`, les opérations sur les polynômes qui dépendent des rationnels : multiplication d'un polynôme par un rationnel, injections des rationnels dans les polynômes (constants)...
  - Dans `Bern`, les définitions des opérations de calcul sur les coefficients de Bernstein
  - Dans `Alg`, les définitions des opérations de calcul sur les codages triangulaires algébriques.

4.2.3 Partage dans la structure d'échantillon cylindrique

Dans la section 4.2.1 on a vu que la structure d'arbre qui représente l'échantillon cylindrique de points est un arbre qui grandit par les feuilles. En effet, pour calculer

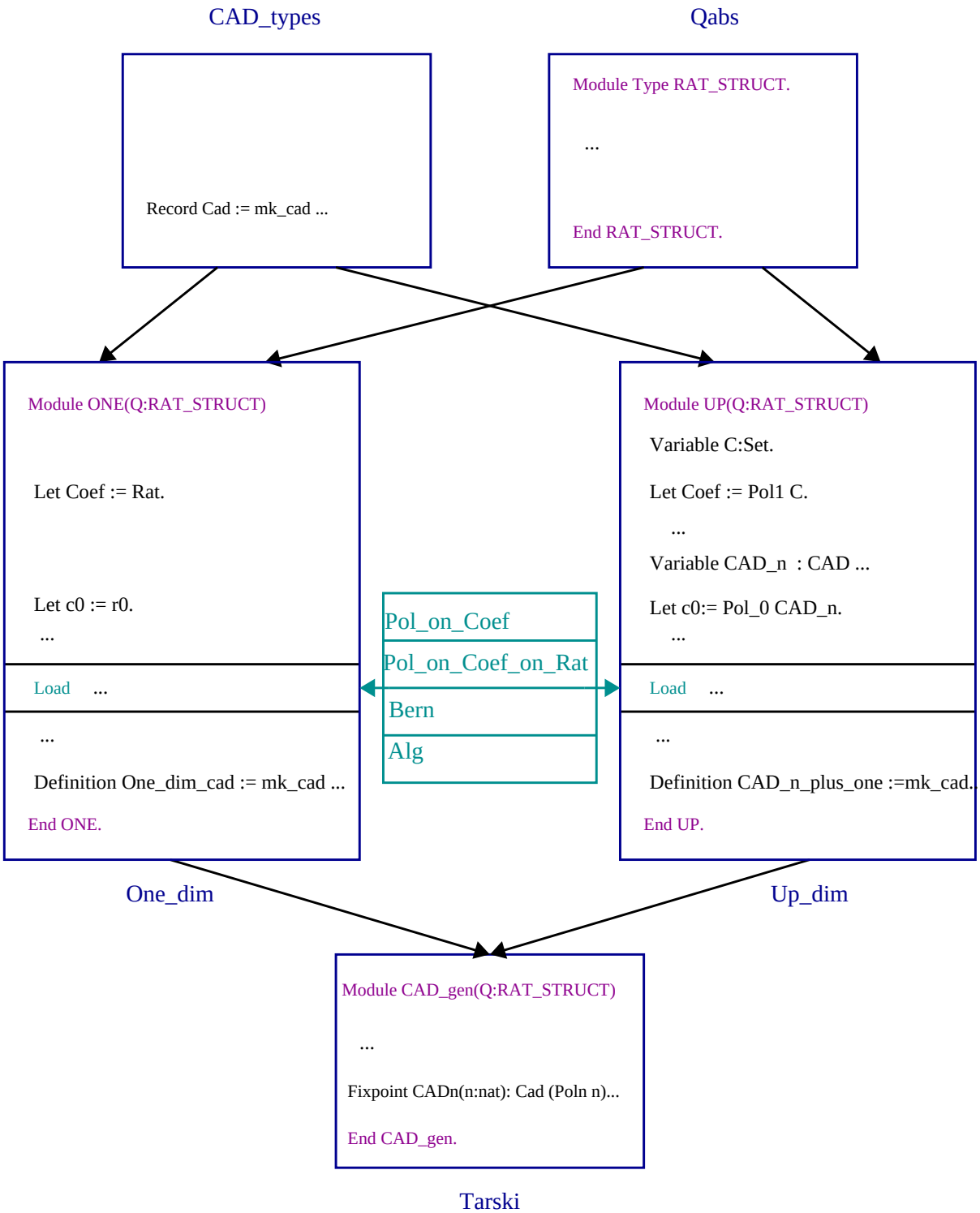


FIG. 4.3: *Architecture du développement*

un échantillon de niveau supérieur, on remplace une feuille d'un arbre par une liste de nouveaux noeuds (voir la figure 4.2).

En calculant la liste des nouveaux noeuds qui doivent remplacer une feuille donnée, on peut avoir à calculer des mises à jour des étiquettes des noeuds internes sur le chemin qui mène à cette feuille. Ceci correspond au phénomène de raffinement des codages triangulaires décrit dans la section 3.3.2, il peut se produire dès que l'on calcule une borne de Cauchy ou bien le signe d'un nombre algébrique. Dans la structure arborescente d'un échantillon cylindrique, un chemin de la racine à un nœud correspond exactement à un codage algébrique.

Il est a priori intéressant de propager les calculs de raffinements dans la structure d'arbre, c'est à dire de mettre à jour les noeuds internes de l'arbre au fur et à mesure qu'on les calcule pour éviter la perte d'information sur la précision des encadrements et une répétition inutile des calculs.

La mise à jour dans l'arbre se fait par recopie successive des chemins dont on a calculé un raffinement.

Toutes les opérations qui peuvent conduire à une mise à jour des codages triangulaires renvoie, en plus de leur valeur, la valeur de la mise à jour du codage. C'est le cas par exemple de la fonction `Pol_low_sign` de l'enregistrement décrit sur le code 4.4. En bout de chaîne, l'opération qui transforme une feuille d'une échantillon de niveau n en le nouveau bourgeon (voir encore en figure 4.2), calcule non seulement la liste des nouveaux noeuds, mais aussi une mise à jour des étiquettes du chemin menant à cette feuille.

On suppose que l'on dispose d'un opérateur qui permet la mise à jour d'un échantillon cylindrique de niveau n.

```
Variable path : Set.
Variable mkCad_map : ∀ C D : Set,
    (path → C → (path * D)) → mkCad C → mkCad D.
```

La figure 4.4 illustre le type de cet opérateur `mkCad_map`, en représentant graphiquement les instanciations des types `C` et `D` qui seront faites en pratique.

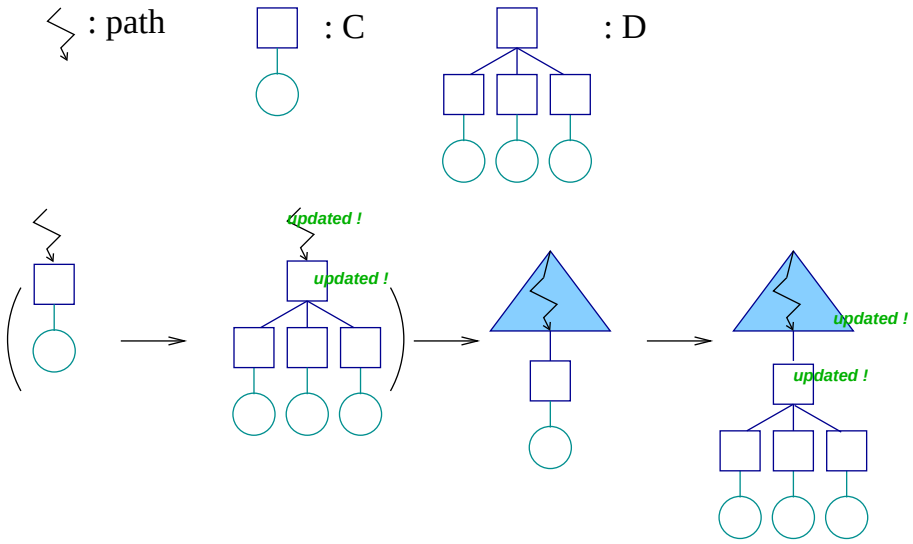


FIG. 4.4: *Construction d'un échantillon de dimension* $n+1$

Le type `path` est celui des chemins dans un échantillon cylindrique de niveau n (donc des codages triangulaires de dimension n). Le type `C` est le type des feuilles dans un échantillon cylindrique de niveau n, et `D` est le type des éléments par lesquels on remplace ces feuilles pour obtenir un échantillon cylindrique de niveau $n+1$.

On dispose d'une opération qui calcule à partir d' une feuille de type `C` et de son chemin de type `path`, une valeur mise à jour du chemin et un élément de type `D`.

L'opérateur de mise à jour, à partir de ces arguments et d'un échantillon cylindrique de niveau n, dont les feuilles sont de type `C`, calcule un échantillon cylindrique de niveau n dont les feuilles sont de type `D`, c'est à dire dans la pratique un échantillon cylindrique de niveau $n+1$. Dans ce nouvel échantillon, les noeuds internes ont été mis à jour.

Dans ces conditions, on peut programmer l'opérateur de mise à jour correspondant au niveau $n+1$, qui propage les raffinements de gauche à droite grâce à la fonction auxiliaire `two_fold`.

```
Variable mkCad:Set → Set.
Definition next_label := mkRpoint.

Definition next_mkCad(C:Set):= mkCad (list (next_label * C)).

Definition next_path := (path * next_label).

Fixpoint two_fold(A B C:Set)
   (f: A → B  → (A*C))(a:A)(l:list B){struct l}: A * (list C):=
   match l with
     |nil ⇒ (a, nil)
     |b::l1 ⇒
       let (a1, c1) := f a b  in
       let (a2, l2) := two_fold f a1 l1 in
         (a2, (c1::l2))
   end.

Definition next_mkCad_map(C D:Set)
(f:next_path → C →  (next_path *  D))
(t:next_mkCad C) : next_mkCad D :=
let aux(p : path)(leave : list (next_label * C)) :
  path*(list (next_label * D)):=
  let f_aux (p:path)(np:next_label*C):path*(next_label*D):=
    let (nlabel,c) := np in
    let (updated_npath,d):=f (p,nlabel) c in
      (fst updated_npath,(snd updated_npath,d)) in
  two_fold path (next_label*C) (next_label * D) f_aux p leave in
  mkCad_map (list (next_label * C)) (list (next_label * D)) aux t.
```

Code 4.5: *Mise à jour dans la structure d'échantillon cylindrique*

Pour finir, on programme le point fixe qui itère la construction de cet opérateur pour toute valeur de n. Ce point fixe est programmé comme dans le code 4.6. La construction **`match ... return ... with`** est celle qui permet au système `Coq` de typer des termes construits par un filtrage, ici sur `n`, dont le type des branches dépend de `n`.

```
Fixpoint mkCad_mapn(n:nat)(C D:Set){struct n}:
  ((cell_pointn n)→ C → ((cell_pointn n) * D))→
  mkCadn n C → mkCadn n D:=
     match n return
       ((cell_pointn n)→ C → ((cell_pointn n) * D))→
           mkCadn n C → mkCadn n D
     with
       |O ⇒
         let res(f : unit → C → unit * D)(x:C):D:=snd (f tt x) in
           fun f : unit → C → unit * D ⇒ (fun x:C ⇒ res f x)
       |S n ⇒
         fun
         (f : (cell_pointn (S n))→ C → ((cell_pointn (S n)) * D)) ⇒
         next_mkCad_map
           Rat (Poln n) (Infon n) (cell_pointn n)
             (mkCadn n) (mkCad_mapn n) C D f
     end.
```

Code 4.6: *Point fixe construisant l'opérateur de mise à jour des échantillons cylindriques*

Comme les types des étiquettes des noeuds sont différents selon leur profondeur dans l'arbre, il est difficile d'écrire un type générique pour les (sous-)arbres manipulés. Ceci serait nécessaire pour utiliser des structures de données plus efficaces(Huet 1997) pour mettre en place ce partage.

D'autre part, même s'il est naturel de partager les mises à jour dans les noeuds internes, sur les exemples que j'ai pu tester, le gain est très faible par rapport à la structure naïve d'arbre non étiqueté dans laquelle les chemins sont stockés dans les feuilles. Dans cette structure naïve, deux feuilles partageant des ancêtres communs ne peuvent tirer parti de cette propriété.

Le faible impact de cette optimisation s'explique en grande partie par la faible complexité géométrique des problèmes résolus en temps raisonnable par ce programme : le nombre de variables est faible, ce qui limite la profondeur des arbres, et le nombre de cellules au dessus d'un même point est souvent peu élevé.

4.3 Un prototype de tactique

4.3.1 L'approche réflexive

L'objectif de ce travail est de faire de l'algorithme de calcul de CAD que nous avons implémenté dans le système Coq le moteur d'une tactique réflexive (voir section 1.2.1) pour l'assistant à la preuve.

Rappelons que la programmation d'une tactique réflexive comprend trois grandes phases :

- La première phase consiste à *définir la syntaxe* de la classe de problèmes à laquelle on s'intéresse. Pour cela, on définit un type inductif, sur lequel on pourra donc programmer par filtrage, qui décrit cette syntaxe. On doit également fournir une

fonction d'interprétation qui transforme un élément du type abstrait en une instance concrète du problème dont on veut automatiser la preuve. Enfin, un oracle programmé dans le méta-langage du système doit permettre de "deviner" la version abstraite, syntaxifiée d'une instance de ce problème.
- La seconde phase est la *programmation de la procédure de décision* à proprement parler, qui est une fonction prenant en argument un élément du type abstrait défini dans la première étape et qui calcule une valeur booléenne.
- Enfin, la dernière phase doit fournir le *théorème de correction* de la procédure. On doit construire la preuve formelle qu'une réponse positive de la procédure de décision sur un argument donné constitue effectivement une preuve de l'interprétation concrète de cet argument.

Ici la classe des problèmes dont on veut *réfléchir* la syntaxe est l'ensemble des formules closes du premier ordre de la théorie de corps réel clos archimédien. On construit donc un type abstrait pour les formules du premier ordre dont les atomes sont des conditions de signes sur des polynômes à coefficients rationnels.

```
Inductive Atom : Set :=
  | Pos : Pol → Atom
  | Neg : Pol → Atom
  | Zero : Pol → Atom.
```

```
Inductive Form : Set :=
 | Aatom   : Atom → Form
 | Anot    : Form → Form
 | Aand    : Form → Form →Form
 | Aor     : Form → Form →Form
 | Aall : Form → Form
 | Aex : Form → Form.
```

FIG. 4.5: *Définition inductive du type abstrait des formules*

La définition de la figure 4.5 suppose implicitement que l'on a fixé le nombre de variables et un ordre sur ces variables.

On définit aussi l'interprétation intuitive `Phi : Form → Prop` qui transforme une formule abstraite en une proposition concrète (qui est un terme Coq de type la sorte **Prop**). Cette interprétation se définit simplement par induction sur la structure de la formule abstraite. Chaque atome abstrait est envoyé vers sa version concrète grâce à l'évaluation des polynômes en une liste de variables réelles. Chaque connecteur et chaque quantificateur abstrait est transformé en son analogue concret dans **Prop**.

Pour programmer l'oracle qui associe à chaque formule concrète (élément de la sorte **Prop**) de la théorie du premier ordre des nombres réels un élément de type `Form`, on dispose dans le système Coq de deux outils. Le premier est le métalangage Ltac, utilisable en ligne de commande, que nous avons déjà utilisé dans le chapitre 1 lors de la conception de la tactique d'automatisation dans les structures d'anneau. Malheureusement, l'expressivité de ce métalangage est limitée. En particulier, il ne permet pas à ce jour d'accéder au nom d'une variable liée par un quantificateur dans le but traité par la tactique. Ceci est un obstacle rédhibitoire à l'utilisation de Ltac pour la fabrication de la tactique générale. En revanche, il est possible d'écrire un prototype de tactique en Ltac qui ne résout que les formules universellement quantifiées. Pour ces formules, l'introduction systématique des quantificateurs universels permet de récupérer les noms des variables qui sont désormais devenues des éléments du contexte. Pour écrire l'oracle, et par suite la tactique, le plus

général, il faudra utiliser un programme Ocaml, qui est le langage dans lequel Coq est lui-même écrit.

Il faut enfin programmer la procédure de décision f qui calcule une valeur de vérité booléenne pour chaque élément de type `Form`. Dans le cas qui nous occupe, cette procédure de décision est celle qui est décrite dans la section 3.1.4, et dans le code 3.4. On rappelle que cette procédure extrait tout d'abord de la formule à décider la famille des polynômes figurant dans ses atomes. Puis, elle calcule une CAD de cette famille (ou plus précisément un échantillon cylindrique de points). Finalement la procédure de décision calcule la valeur de vérité de la formule de départ en parcourant l'arbre calculé par la procédure de CAD en fonction de sa structure.

Si on note `f : Form → bool` la procédure de décision qui à une formule de type `Form` associe sa valeur de vérité en tant que formule de l'arithmétique du premier ordre sur les réels, le théorème de correction de la tactique s'énonce ainsi :

```
Theorem f_correct : ∀ F:Form, (f F) = true ⇒ (Phi F)
```

Lorsque l'utilisateur utilise cette tactique pour obtenir une preuve d'une formule, l'oracle calcule d'abord un élément `F : Form` tel que cette formule soit syntaxiquement égale à `(Phi F)`. Puis la tactique applique le théorème `f_correct` ci-dessus, pour cette formule abstraite `F`. Enfin, le vérificateur de types calcule la valeur de `(f F)`, c'est à dire de la procédure de décision appliquée à `F`. Si cette valeur est bien `true`, alors la preuve est terminée. Sinon, la tactique échoue.

La preuve du théorème de correction de la procédure est bien sûr un défi important. Cette seule preuve utilise en fait toutes les preuves de correction de toutes les fonctions utilisées dans le calcul de la CAD, depuis les opérations d'arithmétique polynomiale jusqu'à la correction de l'opérateur d'élimination des variables. Ce théorème, dont la preuve formelle est très grosse, est établi (et vérifié) une fois pour toute, ensuite son énoncé sera utilisé à chaque application réussie de la tactique.

4.3.2 Prototype et exemples d'utilisation

Pour le moment, seule est disponible une version prototype de la tactique nommée `Tarski`, qui traite les buts quantifiés universellement. Le calcul effectué utilise en effet notre programme Coq de calcul d'échantillon cylindrique, mais le théorème de correction de la tactique est posé en axiome. Comme la preuve de ce théorème n'influe pas sur les performances de la tactique en termes de temps de calcul, ce prototype donne une idée exacte des temps de preuve en utilisant cette méthode, indépendamment de l'état d'avancement de la preuve formelle de la correction.

Les exemples ci-après ont pour but de montrer les problèmes que l'on peut envisager de traiter de façon automatisée avec une tactique basée sur l'algorithme de CAD. Les temps d'exécution donnés représentent le calcul d'une CAD complète pour la famille de polynômes considérées. Pour la taille des problèmes envisagés, le calcul de la forme syntaxifée et le parcours de l'arbre lors de la phase de décision se font en temps négligeable. L'exécution est réalisée par le système Coq en utilisant la réduction compilée vers une machine virtuelle (Grégoire 2003) et une implémentation des nombres rationnels construite au dessus de l'arithmétique modulaire certifiée des entiers récemment implémentée en Coq (Grégoire et Théry 2006). Le processeur utilisé est un Pentium 4 à 3GHz.

Un polynôme, une variable, des racines simples

Calcul d'une CAD pour la famille réduite à un unique polynôme :

$$\{(X-1)(X-2)\dots(X-n)\}$$

n	8	10	12	14	16	18	20
secondes	17	80	349	1057	3167	7556	16885

Remarquons que jusqu'à $n = 20$, le calcul de la CAD pour la famille à n polynômes

$$\{(X-1),\dots,(X-n)\}$$

est immédiat.

Deux polynômes, une variable, des racines communes et des racines propres

Calcul d'une CAD pour la famille à deux polynômes :

$$\{(X-1)(X-2)\dots(X-2n),=(X-n)(X-(n+1))\dots(X-3n)\}$$

Ces deux polynômes sont de degré $2n$ et ont n racines communes et n racines propres.

n	4	5	6	7	16	18	20
secondes	18	86	370	1138	3376	8063	18031

Avec des cercles

Deux cercles disjoints : $\{(X-4)^2+Y^2-1, X^2+Y^2-1\}$
Calcul en 17 secondes
Deux cercles tangents : $\{X^2+Y^2-1, (X-2)^2+Y^2-1\}$
Calcul en 27 secondes
Deux cercles qui s'intersectent : $\{(X-1)^2+Y^2-1, X^2+Y^2-1\}$
Calcul en 17 secondes

Trinôme du second degré

La décomposition algébrique cylindrique du polynôme en quatre variables aX^2+bX+c est calculée en un peu moins de 3 minutes.

Un exemple de J. Avigad

Dans son rapport sur la formalisation en *Isabelle* du théorème de nombres premiers (Avigad, Donnelly, Gray, et Raff 2005), J. Avigad donne un exemple type des problèmes de manque d'automatisation que les auteurs ont rencontré au cours de leur travail de formalisation. Ce problème en quatre variables, de degré total 3 mais linéaire en chacune des variables est traité en 156 secondes par la tactique.

Le théorème de Feuerbach

Le théorème de Feuerbach est un résultat de géométrie du triangle qui établit que dans une triangle, le cercle d'Euler est tangent aux trois cercles exinscrits et au cercle inscrit. Un système de coordonnées cartésiennes bien choisi permet d'exprimer ce problème sous la forme d'un système d'inégalités polynomiales :

$\forall xy, x*(-x+x^2+y^2) > 0 \wedge y^2x(x-1) > 0 \wedge x > 0 \wedge y > 0 \wedge -((-4)x^7 + x^8 - 4 * xy^4 * (-31+y^2) - 4 * x^5 * (1+3*y^2) - 4x^3y^2 * (2+3*y^2) + x^6(6+4y^2) + y^4(-63 + 2y^2 + y^4) + x^4(1+14y^2+6y^4) + 2x^2y^2 - 27y^4 + 2y^6) \leq 0$

Ce problème à deux variables et de degré total 8 n'est pas calculé en temps raisonnable par le programme dans son état actuel.

4.4 Conclusion et prolongements

La procédure de calcul d'un échantillon cylindrique exhaustif pour une famille quelconque de polynômes ayant un nombre de variables arbitraires est complètement achevée dans le système Coq. Ce travail repose sur une architecture de code assez élaborée. L'utilisation de Coq comme langage de programmation impose certains idiomatismes, notamment comme nous l'avions déjà mentionné au chapitre 1 dans l'écriture des points fixes. Néanmoins ceci ne constitue pas, et de loin, la difficulté majeure de cette implantation. Nous pensons que le travail n'est pas significativement plus difficile que si l'on avait choisi un langage de programmation plus conventionnel comme Ocaml. Les types dépendants du language de programmation de Coq ont par ailleurs permis d'obtenir une solution élégante pour la mise en place de la récursivité de l'algorithme. Concevoir une implémentation complète d'un algorithme de calcul de CAD reste néanmoins un puzzle algorithmique sophistiqué, essentiellement à cause du traitement des cellules construites au dessus d'un point algébrique[3].

À notre connaissance (Brown 2004) il existe au moins quatre implémentations complètes et opérationnelles d'algorithmes de calcul de CAD :

- QEPCAD est un programme interactif écrit en C, basé sur la librairie de calcul formel SACLIB. Il est dû à H. Hong mais a bénéficié d'autres contributeurs parmi lesquels G. E. Collins (Hong 1990)
- QEPCADB est issue de QEPCAD mais a été amélioré et étendu par C.W. Brown. Il est développé en C++ et disponible à l'adresse `http://www.cs.usna.edu/~qepcad` (Brown 2003). Il s'agit certainement de l'implémentation la plus complète disponible à ce jour, tant dans les versions de l'opérateur de projection qu'elle propose que dans les outils d'exploitation de l'information donnée par un calcul de CAD.
- RLCAD est implémentée dans l'extension REDUCE du système de calcul formel REDLOG et disponible à l'adresse `http://www.reduce-algebra.com/`.
- Une implémentation de l'algorithme de CAD a été intégrée à la distribution de Mathematica (Strzebonski 2000)

[3] C.W. Brown, auteur de l'implémentation QEPCADB explique dans (Brown 2004) : "I'd also like to point out that while implementing CAD in its full generality is difficult, implementing CAD for full-dimensional cells only is easy!"

R. Rioboo a également réalisé une implémentation complète d'un algorithme de CAD dans le language Axiom[4], ancêtre du language Aldor.

Ce développement constitue l'une des premières utilisations d'un assistant à la preuve comme langage de programmation pour implémenter un algorithme de calcul formel de cette envergure, tant en termes de taille du code source exécuté qu'en termes de complexité algorithmique.

Ce passage à l'échelle à été rendu possible par les orientations récentes du système vers le calcul, et en particulier grâce à l'introduction d'une réduction compilée (Grégoire 2003). Le choix d'abstraire l'implémentation des coefficients constants des polynômes traités, qui sont aussi les objets du calcul, nous a permis d'utiliser ce programme comme une plate-forme de tests. Ceux-ci ont montré qu'une implémentation efficace des calculs sur les nombres peut augmenter de façon significative la puissance du système par l'automatisation qu'elle rend possible.

Dans les développements antérieurs à ces possibilités de calculs efficaces, des calculs aussi coûteux étaient plutôt délégués à des oracles externes comme un système de calcul formel ou un moteur de calcul programmé dans un autre langage de programmation (souvent Ocaml dans le cas du système Coq). Avoir recours à un oracle externe qui communique avec un système de preuves formelles impose de pouvoir reconstruire à l'intérieur du système une preuve formelle de chaque réponse de l'oracle. Ce mode opératoire est donc très adapté aux problèmes dont une solution est difficile à trouver mais facile à démontrer. Un exemple de succès de ce type de mariages est l'utilisation en HOL-Light d'un système expert de programmation semi-linéaire pour une tactique qui montre la positivité d'un polynôme (Harrison 2005). Cette tactique essaie simplement de calculer une identité algébrique décomposant le polynôme candidat en somme de carré. Si le programme échoue, la tactique aussi. Mais s'il réussit, l'identité algébrique est aisément vérifiée et la positivité qui en découle, aussi. Une preuve de ce genre de problème dans un système de preuves formelles sans disposer de la force de calcul de cet oracle s'avère souvent extrêmement pénible, voire impossible. Construire une tactique complète pour la théorie des corps réels clos en délégant les calculs ne semble pas une bonne approche : la recomposition de la preuve de la réponse d'un oracle, comme c'était le cas dans (Mahboubi et Pottier 2002) est dans ce cas un procédé très long et coûteux. Néanmoins ces deux approches de conception de procédures de décisions ne sont pas du tout exclusives. Bien au contraire, elles doivent se compléter pour obtenir des outils très efficaces. On peut en effet imaginer de faire précéder le lancement d'une tactique comme `Tarski`, peu efficace mais complète, par des heuristiques comme celle mentionnée ci-dessus. On rejoindrait en fait de la sorte l'approche des système de calcul formels comme Mathematica, qui n'utilisent le calcul de CAD que lorsque une batterie d'heuristiques ont échoué.

[4]http ://savannah.nongnu.org/projects/axiom/

Chapitre 5

Un principe d'induction pour les nombres réels

Dans ce chapitre, nous proposons une preuve constructive d'un résultat d'analyse réelle appelé *principe d'induction ouverte*. La preuve classique de ce lemme est élémentaire mais le point de vue constructif est rendu délicat par le mélange de topologie et d'analyse réelle qui intervient dans son seul énoncé. La preuve proposée est largement inspirée de celle publiée par Th. Coquand qui illustre le succès de la topologie sans les points pour extraire un contenu calculatoire des arguments du type lemme de König.

5.1 Un lemme trivial d'analyse réelle

L'origine de ce travail est une remarque de G. Dowek à la suite d'un travail de formalisation de notions élémentaires de cinématique pour le traffic aérien (Muñoz, Butler, et Carreño 2001) dans le système PVS. Le principe d'induction ouverte est apparu dans ce travail comme un rouage important des preuves du calcul différentiel élémentaire.

En analyse réelle classique, il est facile de démontrer le lemme suivant :

Proposition 5.1.1 (Induction fermée) *Soit P un ensemble fermé non vide inclus dans $\mathbb{R}$ et c un réel appartenant à P. On suppose que P a la propriété suivante :*

$$\forall x \in [c, +\infty[,\ \exists \varepsilon > 0\ tq\ [x, x+\varepsilon[\subseteq P\ (*)$$

Alors $[c, +\infty[\subseteq P$.

Preuve : Par l'absurde : supposons qu'il existe $y > c$ tel que $y \notin P$. On définit l'ensemble $E := \{x \in \mathbb{R} | x \leq y\ et\ x \in P\}$. E est fermé et majoré : il a un plus grand élément M.

Par définition de E, $M \in P$. Alors d'après $(*)$, il existe $\varepsilon > 0$ tel que $[M, M+\varepsilon[\subseteq P$, ce qui contredit la maximalité de M. □

L'appellation «induction fermée» provient de l'analogie avec les principes d'induction usuels, comme la récurrence sur les entiers naturels. En effet on se donne dans les deux cas un sous-ensemble non-vide (étape d'initialisation) d'un ensemble X, possédant une certaine propriété d'hérédité (hypothèse d'induction) et on montre que cet ensemble vaut X tout entier.

L'énoncé de ce résultat ressemble également beaucoup à celui d'une propriété de connexité : un ensemble à la fois fermé et «ouvert à droite» recouvre finalement un

intervalle tout entier. De fait, ce lemme est un argument standard qui permet par exemple d'établir l'existence d'une unique solution maximale au problème de Cauchy dans le théorème de Cauchy-Lipschitz.

Il est possible d'échanger les rôles des propriétés de fermé et d'ouvert pour obtenir le résultat suivant :

Proposition 5.1.2 (Induction ouverte) *Soit A un ouvert du segment $[0,1]$, possédant la propriété suivante :*

$$\forall x \in [0,1], \quad [\forall y < x \quad y \in A] \Rightarrow x \in A \ (**)$$

Alors $A = [0,1]$.

Preuve : Par l'absurde : supposons qu'il existe $y \in [0,1]$ tel que $y \notin A$. On définit l'ensemble $E := \{x \in [0,1] | x \notin A\}$. E est fermé, non vide, minoré : il a un plus petit élément m.

Par définition de E, $m \notin A$ et la minimalité de m implique que $\forall x \in [0, m[, x \in A$. Alors $(**)$ force $m \in A$ ce qui est contradictoire. □

On peut également montrer ce dernier résultat, toujours par l'absurde, en utilisant la caractérisation des ouverts de $\mathbb{R}$ comme réunion dénombrable d'intervalles ouverts disjoints.

Cette preuve se transpose directement au cas où A est un ouvert d'un segment réel quelconque $[a,b]$. Puis, si A est un ouvert de la demi-droite réelle $[c,+\infty[$, on montre que A recouvre tout segment inclus dans cette demi-droite pour prouver enfin que A vaut en fait $[c,+\infty[$ tout entier.

Finalement, on obtient le corollaire :

Corollaire 5.1.1 *Soit A un ouvert de la demi-droite $[c,+\infty[$. Si A possède la propriété $(**)$, alors A vaut $[c,+\infty[$ tout entier.*

Ce principe d'induction ouverte, précisément dans sa formulation sur $[0,1]$ n'est pas sans rappeler le problème de la course d'Achille contre la tortue[1]. Achille partant du point de départ 0 cherche à atteindre le point d'arrivée 1 en courant. Sa course vérifie deux propriétés :

- Achille atteint un point p du parcours si et seulement si il a atteint tous les autres points situés strictement entre le départ et p.
- Si Achille atteint un point p situé strictement entre le départ et l'arrivée, alors il ira un peu plus loin, et atteindra un point $p' > p$.

La question est donc la suivante : est-ce que ces deux conditions suffisent à assurer Achille qu'il arrivera à destination ?...

Les preuves que nous avons données de ces deux lemmes sont bien sûr des preuves classiques, puisque elle reposent toutes deux sur une utilisation directe de l'axiome de tiers-exclus. On assure Achille que s'il n'arrivait pas à bon port, on se trouverait dans une situation absurde. Par cet artifice, on cache complètement le *contenu calculatoire* de ces énoncés. Lorsqu'on utilise le principe de récurrence sur les entiers pour démontrer un prédicat de la forme $\forall n P(n)$, on peut comprendre le calcul effectué pour prouver une instance de ce prédicat, i.e. $P(n_0)$ pour un $n_0 \in \mathbb{N}$ particulier, en éliminant les «coupures

[1] Cette analogie s'inspire de (Veldman 2001).

de récurrence» dans la preuve par induction qui se termine par une élimination du quantificateur universel $\forall$. On prouve alors successivement et sans utilisation du schéma de récurrence : $P(0), P(1), \ldots, P(n_0)$. C'est ce qu'on appelle le contenu calculatoire de la récurrence sur les entiers naturels. Dans le contexte qui nous occupe, en partant d'une preuve classique comme celle que l'on a proposé ci-dessus pour le lemme d'induction ouverte, il est impossible d'expliciter de la sorte la construction de la preuve qu'étant donné un point quelconque de $[0, 1]$, Achille passe bien par ce point. Pour ce faire, il faudra obtenir une preuve *constructive* de l'énoncé, que l'on sache interpréter comme un calcul.

5.2 Preuves constructives

5.2.1 Contenu calculatoire et constructivisme

L'origine de l'intuitionnisme remonte à la thèse de doctorat de L. E. J. Brouwer en 1907. Il s'agit pour Brouwer de contribuer au débat alors virulent sur les fondations des mathématiques, et de proposer une description précise de sa conception de l'activité mathématique. Il formule explicitement la séparation essentielle qui doit être faite à son sens entre le contenu mathématique, réceptacle de l'intuition, et le langage mathématique, consistant en une manipulation de symboles régie par des règles de syntaxe, et qui prendra plus tard le nom de meta-mathématique. Les constructions mathématiques ont dans ce contexte une existence autonome et une validité intrinsèque, le langage (méta-)mathématique est quant à lui potentiellement imparfait et son rôle est cantonné à la description de l'édifice mathématique, ne contribuant en aucun cas à l'augmenter ou le modifier. Un des attributs majeurs de cette vision des mathématiques est la remise en cause du principe du tiers-exclus. De fait, tout preuve dans le système intuitionniste contient un procédé algorithmique qui construit un témoin de l'énoncé validé. Notons qu'historiquement, il n'est pas question de préciser quel est le modèle de calcul dans lequel on place ces programmes qui construisent les témoins.

Au delà de l'aspect constructif de telles preuves, qui résultent en calculs sur les objets mathématiques du raisonnement, le point de vue constructif implique un changement radical dans la façon d'envisager la notion de preuve. L'intuitionnisme décentre l'objet de la logique en privilégiant, plutôt que la notion de valeur de vérité qui était au coeur de la sémantique des preuves en logique classique, celle de causalité. On peut observe ce phénomène sur le simple énoncé $A \Rightarrow B$. En logique classique, cette implication peut être lue comme le fait que "dès que A est vraie alors B est vraie". D'un point de vue intuitionniste, on interprète une preuve de l´implication comme la donnée d'un procédé systématique de transformation de toute preuve de l'énoncé A en une preuve de l'énoncé B. Il s'agit d'une remise en question complète de la prouvabilité classique, motivée par une critique fondamentales des démonstrations classiques : qu'a-t-on prouvé quand on donne une preuve de $A \vee B$? les quantificateurs $\exists$ et $\forall$ induisent-il l'existence de fonctions calculables ?

L'interprétation de Brouwer-Heyting-Kolmogorov (BHK) permet de préciser le modèle de calcul qui décrit les constructions des preuves de la logique intuitionniste, sous la forme de programmes fonctionnels. L'isomorphisme de Curry-de Bruijn-Howard étend et précise cette correspondance en interprétant les programmes comme des termes d'un λ-calcul typé. Le point de vue réciproque sur la correspondance de BHK incite à s'intéresser

aux classes de programmes fonctionnels typés par une même formule logique. T. Griffin (Griffin 1990) découvre que l'on peut typer certaines instructions standard des langages de programmation fonctionnels par des formules classiques, c'est à dire non prouvables au sens intuitionniste.

La constructivité n'est ainsi pas l'apanage de la logique intuitionniste, et il faut nuancer l'opposition classique/constructif. Cette étude réciproque trouve en fait son cadre formel dans l'étude la réalisabilité. La réalisabilité standard est une technique introduite par S. Kleene pour construire des modèles combinatoires de la logique intuitionniste. Cette technique se prolonge selon les grandes lignes suivantes : étant donné un langage de programmation fonctionnel, qui est choisi selon les besoins du développement (λ-calcul, fonctions récursives, Lisp,...), et une théorie, on se donne des règles d'interprétation qui associent à chaque formule de la théorie, un ensemble de programmes. Cet ensemble de règles est en fait une adaptation de l'interprétation de BHK au langage de programmation choisi. Puis on s'attache à montrer que si une formule est prouvable dans la théorie, alors elle est réalisable, en général par une induction sur la structure des dérivations de preuves. En fait la difficulté principale repose sur la réalisabilité des axiomes de la théorie. Cette technique possède un champ d'application très vaste : la réalisabilité classique à la Krivine permet d'étendre l'interprétation de BHK à la logique classique toute entière, en utilisant un langage de programmation avec continuations, il existe également une lignée de travaux portant sur la réalisabilité de la théorie des ensembles.

5.2.2 Topologie, analyse réelle et extraction de programmes

Le problème qui nous occupe se situe à la croisée de deux domaines mathématiques radicalement modifiées par le point de vue constructif, à savoir la topologie et l'analyse réelle. L'un des plus éminents architectes de la "reconstruction" des mathématiques décrit ainsi ce phénomène : "Very little is left of general topology after that vehicle of classical mathematics has been taken apart and reassembled constructively. With some regret, plus a large measure of relief, we see this flamboyant engine collapse to constructive size" (Bishop 1967). En analyse réelle, le théorème de Bolzano-Weierstrass, par exemple, perd sa validité, une fonction continue n'atteindra ainsi pas ses extrema sur un compact, mais on pourra approximer ceux-ci aussi précisément que désiré. Ces modifications proviennent de la nature infinitaire des objets de l'étude, nature le plus souvent inerte en analyse réelle classique.

La réhabilitation constructive passe par deux étapes indissociables : le choix d'une axiomatique satisfaisante, et la reconstruction des preuves des énoncés éventuellement reformulés. Il existe néanmoins des techniques pour extraire des preuves constructives à partir d'arguments classiques. L'une d'entre elles, la topologie sans les points est particulièrement adaptée aux arguments de compacité combinatoire (Coquand 1991). Elle consiste à modifier la granularité de l'étude et à lui donner comme objet, non plus les points de l'espace topologique, mais ses ouverts. Coquand fait remonter les origines de cette idée aux travaux de Whitehead au début du XXème siècle, et forge à partir de celle-ci un outil pour transformer des énoncés classiques dont la preuve nécessite l'usage de l'axiome du choix en énoncés prouvables constructivement. Cette changement de point de vue s'avère avantageux dans le contexte de la formalisation des preuves en théorie des types car la description d'espaces topologiques "sans les points" se formule assez naturel-

lement à l'aide de définitions inductives.

Les deux preuves constructives de la propriété 5.1.2 de la littérature dont nous avons eu connaissance illustrent deux démarches différentes permettant d'obtenir une preuve constructive, et par suite un sens calculatoire, à un résultat prouvé par un argument classique.

L'une d'elle (Veldman 2001) s'appuie explicitement sur la construction des réels comme suites de Cauchy avec module de convergence. Cette preuve montre que l'on peut améliorer l'argument de compacité combinatoire habituel, qui est le lemme de l'éventail[2] pour obtenir le lemme d'induction ouverte.

La seconde (Coquand 1997) abstrait la définition des points réels pour obtenir une preuve basée sur la notion d'ouverts. Le procédé de calcul de la preuve porte donc lui-même sur les ouverts et non sur les points réels, qui ont disparu du contexte. Cette preuve utilise le principe d'induction barrée qui assure la terminaison de l'énumération d'arbres à branchement potentiellement dénombrable sous certaines conditions (voir section 5.2.3). Cette version de la preuve travaille sur l'ensemble de Cantor, c'est à dire l'ensemble des suites $\{0,1\}^{\mathbb{N}}$.

5.2.3 Arbres et terminaison

Dans toute la suite, on désigne par $\mathbb{N}^*$ l'ensemble des mots finis sur les entiers naturels et par $\mathbb{N}^\omega$ l'ensemble des mots infinis sur les entiers naturels.

Si α est dans $\mathbb{N}^\omega$ et si $x \in \mathbb{N}$, alors $\alpha(x)$ désigne le préfixe de longueur x de α, à savoir le mot $\alpha_0, \ldots, \alpha_{x-1}$.

$*$ désigne la concaténation des mots finis d'entiers et $u \bullet a$ est la notation utilisée pour la liste formée par concaténation de u et du mot a de longueur 1.

Si u est un mot fini, $|u|$ désigne sa longueur.

ε désigne le mot vide.

Du point de vue de la réalisabilité les axiomes présentés dans cette section assurent la terminaison de parcours dans des arbres dont les chemins sont de longueur finie.

Induction barrée

L'axiome d'induction barrée est un axiome concernant les arbres à branchement éventuellement dénombrable, de profondeur éventuellement infinie. En particulier, on peut en dériver trivialement le lemme de l'éventail. C'est Brouwer qui formule, et utilise, le premier une version de cet axiome, dont on trouve plusieurs variantes dans la littérature (Troelstra 1973).

L'ensemble $\mathbb{N}^*$ peut être vu comme (l'ensemble des positions des nœuds dans) l'*arbre universel*, dont les branches, infinies, sont les éléments de $\mathbb{N}^\omega$. Une *barre* est un ensemble de nœuds de cet arbre qui ne peut être évité quand on parcourt un chemin infini (voir figure 5.1.

Définition 5.2.1 *Barres*

[2] Il s'agit de la version intuitionniste du lemme de König, à savoir : tout arbre à branchement fini dont les chemins sont de longueur finie est constitué d'un nombre fini de nœuds

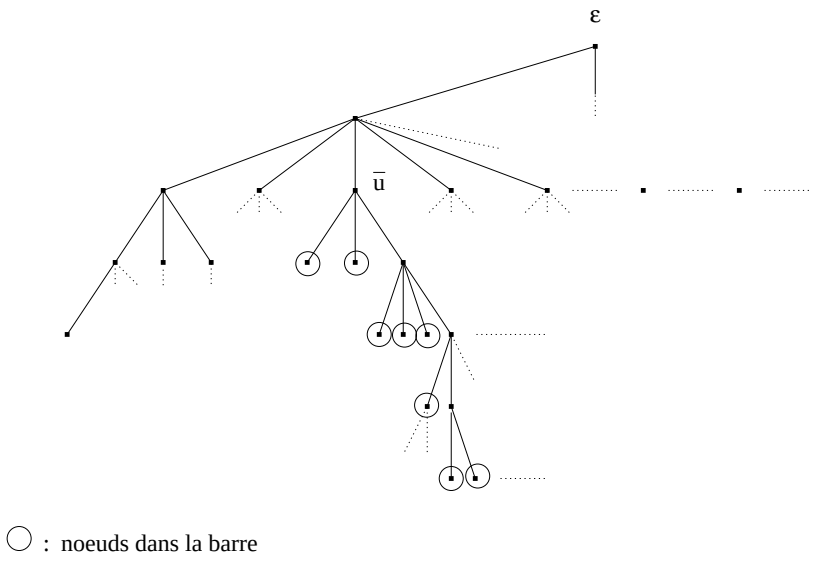


FIG. 5.1: *Un barre pour le nœud u*

Soit X un prédicat sur $\mathbb{N}^*$ et u un mot de $\mathbb{N}^*$. On dit que X barre le mot u, noté $X|u$, si pour tout chemin infini α qui admet u comme préfixe, il existe $k \in \mathbb{N}$, tel que $X(\alpha(k))$.

Si E est un ensemble de mots finis, on dira que X barre E s'il barre tous les éléments de E.

On dira que X est une barre si X barre ε.

Dans ce qui suit, on identifie un nœud de l'arbre universel avec le mot fini de $\mathbb{N}^*$ qui code sa position. Étant donné un prédicat Y sur les mots finis, i.e. un ensemble de positions de nœuds dans l'arbre infini $\mathbb{N}^\omega$, le principe d'induction barrée permet d'assurer sous certaines hypothèses que cet ensemble de nœuds Y contient bien la racine ε.

Considérons deux ensembles de nœuds X et Y, tels que Y contienne X et qui vérifient les hypothèses suivantes :

- Y est *héréditaire* : c'est à dire que si tous les fils d'un nœud sont dans Y, on a aussi montré que ce nœud lui-même est dans Y.
- X est *monotone* : c'est à dire que si un nœud est dans X, tous les descendants de ce nœud le seront aussi.
- X est *une barre* : si l'on a parcouru dans l'arbre un chemin fini qui nous a mené de la racine à la position u, quelle que soit la façon dont on prolonge ce chemin, un nœud au moins du chemin infini sera dans X.

Sous ces hypothèses, le principe d'induction barrée assure que tous les nœuds sont dans Y : un chemin qui passe par un nœud u coupe forcément X. Si ce chemin rencontre X en un préfixe de u, on conclut par monotonie de Y que u est dans Y. Sinon, il faut utiliser la propriété d'hérédité et initier un procédé récursif dont l'axiome d'induction barrée assure la terminaison.

On peut énoncer l'axiome d'induction barrée sous la forme de la règle d'inférence suivante :

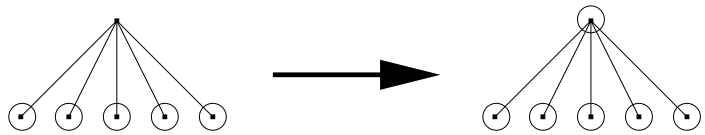


FIG. 5.2: *Hérédité*

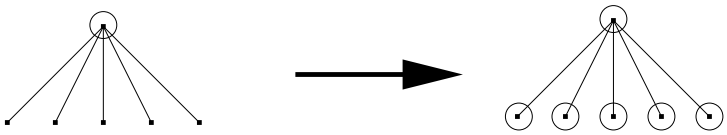


FIG. 5.3: *Monotonie*

Axiome d'induction barrée :

$$\frac{\begin{array}{ll} \forall u \in \mathbb{N}^* \ [X(u) \Rightarrow Y(u)] & X\text{est inclus dans } Y \ (1) \\ \forall u \in \mathbb{N}^* \ \forall a \in \mathbb{N} \ [X(u) \Rightarrow X(u \bullet a)] & X\text{est monotone } (2) \\ \forall u \in \mathbb{N}^* \ [\forall a \in \mathbb{N} \ Y(u \bullet a)] \Rightarrow Y(u) & Y \text{ est héréditaire } (3) \\ \forall \sigma \in \mathbb{N}^\omega \ \exists k \mid X(\sigma_1, \ldots, \sigma_k) & X\text{est une barre } (4) \end{array}}{\forall u \in \mathbb{N}^*, Y(u)}$$

Le point de vue de Coquand est de considérer une formulation inductive de cet axiome, dont les règles de constructions contrôlent la profondeur à laquelle on rencontre la barre sous un nœud.

Définition 5.2.2 (Barres inductives) *Soit X un prédicat sur $\mathbb{N}^*$. Le prédicat $X|_{ind}u$ est défini inductivement par les règles d'introduction suivantes :*

$$\frac{X(u)}{X|_{ind}\ u} \qquad \frac{X|_{ind}\ u}{X|_{ind}\ u \bullet a} \qquad \frac{\forall a \in \mathbb{N},\ [X|_{ind}\ u \bullet a]}{X|_{ind}\ u}$$

On dit que X est une barre inductive si $X|_{ind}\ \varepsilon$.

Le principe d'induction barrée est alors obtenu comme le principe d'induction associé à cette définition inductive. Ce point de vue permet notamment de donner une interprétation du lemme de l'éventail en théorie des types (Fridlender 1999).

Veldman reprend dans (Veldman 2001) l'axiome dans sa formulation donnée par Brouwer pour prouver directement le lemme de l'éventail sans utiliser d'induction barrée. Il semble s'être plutôt intéressé à la prouvabilité du lemme d'induction ouverte avec un axiome finitaire comme celui de l'éventail. En fait il utilise une version *presque-finie* de l'éventail, dans laquelle le branchement des arbres est caractérisé par une notion intuitionnisme de non-infini.

Barres décidables

La formulation de l'induction barrée que nous utilisons dans ce qui suit est une formulation particulière aux arbres dont les chemins sont de longueur finie, proche en fait du lemme de l'éventail. Elle permet de conclure que la racine d'un tel arbre est décidable, sans hypothèse de monotonie mais en disposant d'un prédicat de barre décidable.

Induction barrée avec barre décidable
Soit X et Y deux prédicats sur $\mathbb{N}^*$.

$$\frac{\begin{array}{ll}\forall u \in \mathbb{N}^* \ [X(u) \Rightarrow Y(u)] & X \text{ implique } Y \\ \forall \alpha \in \mathbb{N}^{\mathbb{N}} \ \exists k \mid X(\alpha(k)) & X \text{ est une barre} \\ \forall u P(u) \vee \neg P(u) & X \text{ est décidable} \\ \forall u \in \mathbb{N}^* \ [\forall a \in \mathbb{N} \ Y(u \bullet a)] \Rightarrow Y(u) & Y \text{ est héréditaire}\end{array}}{Y(\varepsilon)}$$

5.3 Induction ouverte pour les ouverts énumératifs

Dans cette section, nous proposons une preuve par induction barrée de la propriété 5.1.2 pour une axiomatique intuitionniste des nombres réels. Il s'agit d'une adaptation de la preuve combinatoire de Coquand qui donne le résultat pour les ouverts de l'ensemble de Cantor (Coquand 1997).

5.3.1 Ouverts énumératifs

On se place dans le cadre d'une axiomatique intuitionniste des nombres réels, comme corps réel clos intuitionniste (voir les axiomes de la définition 3.1.1) archimédien. Les suites de Cauchy intuitionnistes forment un modèle de cette axiomatique.

La topologie dont est munie cette structure réelle est celle des *ouverts énumératifs*.

Définition 5.3.1 (Ouvert énumératif) *Un ensemble ouvert énumératif de $\mathbb{R}$ est une union d'intervalles ouverts à bornes rationnelles donnés par une fonction :*

$$\begin{array}{rccc} g: & \mathbb{N} & \to & \mathbb{Q} \times \mathbb{Q} \\ & n & \mapsto & (\alpha_n, \beta_n) \end{array}$$

Précisément : $A = \bigcup_{n \in \mathbb{N}}]\alpha_n, \beta_n[$

Si E est un sous-ensemble de $\mathbb{R}$, on définit un ouvert énumératif de E comme l'intersection d'un ouvert énumératif de $\mathbb{R}$ avec E, comme il est usuel de le faire pour les topologies induites.

Les ouverts énumératifs ne sont pas nécessairement des ensembles décidables. On prouve qu'un point $x \in \mathbb{R}$ est dans un ouvert dénombrable défini par une fonction $g : \mathbb{N} \to \mathbb{Q} \times \mathbb{Q}$ si on sait calculer un témoin n tel que $g(n) = (\alpha, \beta)$ et $x \in]\alpha, \beta[$.

Le choix de la topologie de $\mathbb{R}$ pour laquelle on établit le résultat est crucial pour la preuve car il est directement lié à la terminaison du calcul effectué au cours de la preuve. Par exemple, définir un ouvert comme la donnée pour chacun de ses points du rayon d'une boule incluse ne serait pas adapté à l'argument utilisé, et d'ailleurs contraire à l'approche consistant à éliminer les points du raisonnement.

5.3.2 Preuve par induction barrée

Le principe d'induction ouverte dont nous proposons une preuve s'énonce ainsi :

Théorème 5.3.1 (Induction ouverte) *Soit A un ouvert énumératif du segment $[0,1]$, possédant la propriété suivante :*

$$\forall x \in [0,1], \quad [\forall y < x \quad y \in A] \Rightarrow x \in A \ (***)$$

Alors $A = [0,1]$.

Preuve Dans ce qui suit, si I est un intervalle fermé borné de la forme $[x,y]$, alors on note I_g l'intervalle fermé $[x, \frac{x+y}{2}]$ et par I_d l'intervalle fermé $[\frac{x+y}{2}, y]$.

On note G la famille des intervalles ouverts qui définissent l'ouvert énumératif A :

$$G := \{]\alpha_i, \beta_i[, i \in \mathbb{N}\}$$

On définit de façon mutuellement récursive le prédicat *acceptable* sur $\mathbb{N}^*$ et une application f qui envoie les éléments de $\mathbb{N}^*$ sur des intervalles fermés de $[0,1]$ à bornes dyadiques :

▶ Définition de *acceptable* : – ϵ est *acceptable*
- $u \bullet 0$ est *acceptable* ssi u est acceptable
- $u \bullet (n+1)$ est *acceptable* ssi $f(u)_g$ est acceptable et si $f(u)_g$ est inclus dans l'union finie d'intervalles $\bigcup_{i<n+1}]\alpha_i, \beta_i[$

▶ Définition de f : – $f(\varepsilon) := [0,1]$
- $f(u \bullet 0) := f(u)_g$
- $f(u \bullet (n+1)) := \begin{cases} f(u)_d \text{ si } u \bullet (n+1) \text{ est } \textit{acceptable} \\ f(u)_g \text{ sinon} \end{cases}$

Remarque 5.3.2.1 *Soit u un élément de $\mathbb{N}^*$ et $a, b \in \mathbb{Q}$ tels que $f(u) = [a,b]$*
- *La longueur de l'intervalle $f(u) = [a,b]$ est $\frac{1}{2^{|u|}}$.*
- *L'intervalle fermé $[0,a]$, éventuellement réduit au singleton $\{0\}$, est inclus dans A, car il est inclus dans une union finie d'intervalles ouverts de la famille G.*

On donne sur la figure 5.4 un exemple possible de configuration pour la fonction f et le prédicat *acceptable*. Un mot fini est représenté par un nœud dont il code la position dans l'arbre universel $\mathcal{T}$. On identifie un nœud avec sa position. Les nœuds acceptables sont encadrés.

Supposons que $[0, \frac{1}{2}]$ est inclus dans $]\alpha_0, \beta_0[\bigcup]\alpha_1, \beta_1[\bigcup]\alpha_2, \beta_2[$ mais pas dans $]\alpha_0, \beta_0[\bigcup]\alpha_1, \beta_1[$. Sur la figure 5.4, certains nœuds de $\mathcal{T}$ sont étiquetés avec les valeurs de f et du prédicat acceptable que l'on peut calculer avec cette information.

Le premier mot de longueur 1 qui soit acceptable est le mot 3, et tout mot n de longueur 1 tel que $n \geq 3$ est *acceptable*, puisqu'on inclut le même intervalle dans une union croissante d'éléments de G.

Le mot 30 de longueur 2 est *acceptable* puisque son préfixe 3 est lui-même *acceptable*. Il faut vérifier si $[\frac{1}{2}, \frac{3}{4}]$ est inclus dans $]\alpha_0, \beta_0[$ ou non pour calculer la valeur de $f(31)$.

Soit X le prédicat sur $\mathbb{N}^*$ défini par :

$$X(u) \stackrel{\text{def}}{\equiv} u \text{ acceptable} \Rightarrow f(u) \text{ est inclus dans } \bigcup_{i<|u|}]\alpha_i, \beta_i[$$

Le prédicat X est décidable. Le lemme suivant fait de X une barre.

Lemme 5.3.2 *Pour toute suite infinie d'entiers $n_1 n_2 \ldots$ de $\mathbb{N}^{\mathbb{N}}$, il existe k tel que $f(n_1 \ldots n_k)$ est inclus dans $\bigcup_{i \leq k}]\alpha_i, \beta_i[$.*

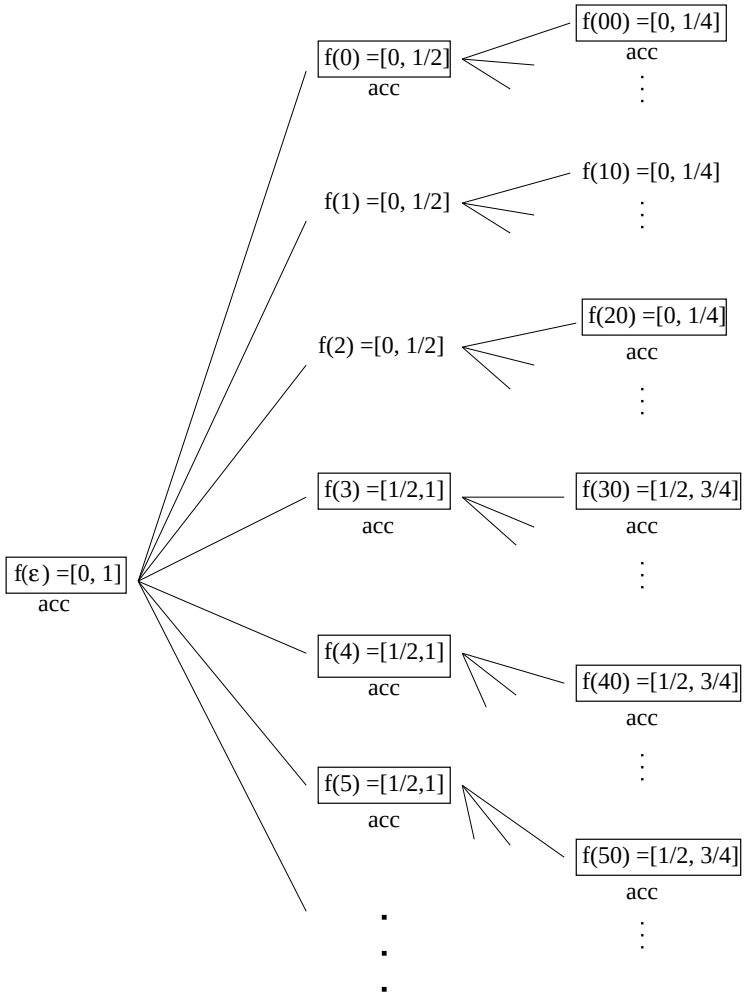


FIG. 5.4: *Exemple de configuration*

Preuve Nous démontrons ici la proposition suivante, qui est plus forte :

Pour toute suite infinie d'entiers $n_1 n_2 \ldots$ de $\mathbb{N}^{\mathbb{N}}$, il existe k tel que $f(n_1 \ldots n_k)$ est inclus dans l'un des intervalles $]\alpha_i, \beta_i[$ for $i = 0 \ldots k$.

Pour tout $k \in \mathbb{N}$, on note $[a_k, b_k]$ la valeur de $f(n_1 \ldots n_k)$. La suite $([a_k, b_k])_{k\in\mathbb{N}}$ est une suite d'intervalles fermés strictement emboîtés, telle que :

$$\forall k \in \mathbb{N}, \quad b_k - a_k = \frac{1}{2^k}$$

La suite $(a_k)_{k\in\mathbb{N}}$ est par suite une de suite Cauchy croissante qui admet l'identité comme module de convergence. Elle converge vers un nombre réel x qui est un élément de A.

D'après la remarque 5.3.2.1, pour tout entier k, l'intervalle $[0, a_k]$ est inclus dans A. Par suite, pour tout $y < x$, $[0, y]$ est aussi inclus dans A. Comme l'ouvert A possède par hypothèse la propriété inductive $(***)$, le point x est aussi un élément de A. A ce titre, il existe un entier n_0 tel que x soit dans l'intervalle $]\alpha_{n_0}, \beta_{n_0}[$ de la famille G. Il existe aussi un entier k_0 tel que l'intervalle $[a_{k_0}, b_{k_0}]$: il suffit de choisir un entier k_0 tel que $\frac{1}{2^{k_0}} \leq \beta_{n_0} - \alpha_{n_0}$.

Soit $k := max(n_0, k_0)$. Comme $k \geq k_0$, on a bien :

$$f(n_1 \ldots n_k) = [a_k, b_k] \subseteq [a_{k_0}, b_{k_0}] \subseteq]\alpha_{n_0}, \beta_{n_0}[\quad \text{avec } n_0 \leq k$$

□

On définit le prédicat Y sur $\mathbb{N}^*$ de la façon suivante :

$$Y(u) \stackrel{\text{def}}{\equiv} u \text{ acceptable} \Rightarrow f(u) \text{ est inclus dans une union finie d'intervalles de } G$$

Pour tout mot fini u, on a $X(u) \Rightarrow Y(u)$. Toutes les conditions sont réunies pour utiliser le principe d'induction barrée avec barre décidable, afin de prouver que l'intervalle $[0, 1]$ est inclus dans une union finie d'éléments de la famille G :

- Pour tout mot fini, $X(u) \Rightarrow Y(u)$.
- X est monotone : pour tout $n \in \mathbb{N}$ et pour tout mot fini $u \in \mathbb{N}^*$, $f(u \bullet n)$ est inclus dans $f(u)$.
- Y est héréditaire : soit u un mot fini acceptable. Supposons de plus que pour tout $n \in \mathbb{N}$, $Y(u \bullet n)$. En particulier, $f(u \bullet 0) = f(u)_g$ est inclus dans une union finie $\bigcup_{i<n+1}]\alpha_i, \beta_i[$. Par définition, $u \bullet (n+1)$ est alors acceptable et $f(u \bullet (n+1)) = f(u)_d$ est inclus dans une union finie d'intervalles de G. Par définition du prédicat *acceptable*, $u \bullet (n+1)$ est alors acceptable, et par hypothèse, $f(u \bullet (n+1)) = f(u)_d$ est par suite inclus dans une union finie d'intervalles de G. L'intervalle $f(u)$ est donc lui aussi inclus dans une union finie d'intervalles de G car c'est le cas pour ses deux moitiés $f(u)_g$ et $f(u)_d$.
- Le lemme 5.3.2 assure que le prédicat décidable X est une barre.

Le principe d'induction barrée avec barre décidable permet d'obtenir $Y(\varepsilon)$ soit :

ε est acceptable $\Rightarrow$ $f(\varepsilon)$ est inclus dans une union finie d'intervalles ouverts éléments de G

Comme par définition ε est acceptable et $f(\varepsilon) = [0, 1]$, ceci prouve que l'ouvert énumératif A muni de sa propriété inductive recouvre le segment $[0, 1]$ tout entier. □

Les interrogations d'Achille semblaient en effet légitimes : si l'objectif d'atteindre le point 1 semble incertain, il en est en fait de même de tous les autres points du parcours qu'on pourrait poser comme jalons : le point $\frac{1}{2}$ peut être atteint en une étape si $g(0)$ contient $[0, \frac{1}{2}]$ mais il faudra peut être beaucoup (infiniment ?) plus d'étapes. Ce point de mi-parcours est en fait un objectif aussi lointain que le point 1. Heureusement, la combinaison des qualités de la course d'Achille avec l'axiome d'induction barrée assurent Achille d'arriver en temps fini à destination.

5.4 Conclusion

La transposition des idées originales de Coquand à une structure réelle munie de la topologie définie par les ouverts énumératifs permet d'obtenir une preuve constructive directe du principe d'induction ouverte pour cette structure.

La preuve précédente (Coquand 1997) établissait le résultat pour l'ensemble de Cantor $\mathcal{K}$, sur lequel on définit les ouverts par des prédicats décidables sur les mots binaires finis. Nous proposons de travailler directement avec des nombres réels, et une topologie qui nous a semblé naturelle. Il nous semble également difficile d'obtenir directement une preuve de l'induction ouverte dans ce cadre, par le biais d'une application continue entre $\mathbb{R}$ et $\mathcal{K}$ par exemple. Par contre, comme le montre la preuve ci-dessus, les idées de la preuve en topologie sans les points sur l'ensemble de Cantor se transposent pour fournir une preuve dans le cadre continu.

Le contenu calculatoire de cette preuve est très simple : la preuve que l'ouvert énumératif A, possédant la propriété $(***)$ recouvre l'espace $[0, 1]$ tout entier est simplement l'énumération des intervalles ouverts à bornes rationnelles qui définissent A. Le principe d'induction barrée vient assurer la terminaison de cette énumération en temps fini, pour obtenir un recouvrement fini de $[0, 1]$ par des intervalles ouverts à bornes rationnelles tous inclus dans A.

Cette exercice d'adaptation laisse plusieurs questions en suspens. La version la plus utilisée du principe en mathématiques classiques semble être l'induction fermée. Du point de vue intuitionniste, les deux prédicats sont très différents, la topologie intuitionniste étant privée de la correspondance duale entre fermés et ouverts. Il faut probablement reconsidérer complètement le problème pour obtenir une preuve constructive de cet énoncé.

Du point de vue du contenu calculatoire, l'axiome utilisé pour assurer la terminaison de la preuve est très puissant, ce qui semble avoir motiver les travaux de Veldman pour circonscrire au mieux l'argument de terminaison requis pour cette preuve. Ce problème est à notre connaissance encore ouvert. Il s'agit là d'un aspect du problème de la classification des axiomes *à la* lemme de l'éventail selon leur puissance de calcul.

Conclusion

Contributions et conclusion

Le travail réalisé dans cette thèse s'articule autour de deux axes.

Le premier est d'apporter au système Coq des outils d'automatisation puissants en exploitant l'efficacité du mécanisme de réduction dont s'est doté le système. Ces évolutions récentes qui placent au premier plan les techniques réflexives pour écrire des tactiques ont été un pré-requis indispensable à ce travail. L'objectif principal est d'apporter une tactique pour l'automatisation de la théorie du premier ordre de l'arithmétique réelle mais le travail conséquent que nécessite ce programme apporte les ingrédients qui permettent la réalisation d'autres outils d'automatisation. Ceux-ci constituent à la fois des contributions à l'amélioration générale de l'environnement de preuve, et des étapes nécessaires pour rendre la preuve de correction de l'algorithme de décision pour les réels réalisable.

Le second est de réaliser une preuve formelle de la correction d'un algorithme de CAD. Ceci représente un défi important compte tenu en particulier de l'absence de bibliothèques de preuves formalisées en Coq qui soient réutilisables dans ce contexte. L'architecture d'une formalisation de cette envergure demande en soi un travail de réflexion sur l'articulation des calculs mis en jeu. Cet aspect que l'on peut qualifier d'"algorithmique des preuves formelles" a influé les choix algorithmiques de programmation pour obtenir une structure de code qui permettent de lui ajuster sa preuve de correction pour aboutir à la tactique réflexive finale.

Dans ces deux axes, on a mené une réflexion pratique et théorique sur la mécanisation des calculs et des preuves en analyse réelle. En effet la réalisation d'une procédure de décision certifiée pour l'arithmétique réelle nécessite une étude transversale, depuis les aspects méta-théoriques de l'axiomatisation des structures logiques, en passant par une compréhension en profondeur des arguments de géométrie semi-algébrique réelle mis en jeu dans l'algorithme de CAD, et finalement un travail algorithmique important pour la mise en œuvre de l'implémentation.

Le travail repose sur les algorithmes proposés par un ouvrage de référence de la géométrie algébrique réelle (Basu, Pollack, et Roy 2003).

Une implémentation complète en Coq de l'algorithme de CAD. La première contribution de ce travail est de proposer une nouvelle implémentation de l'algorithme de décomposition algébrique cylindrique de Collins en utilisant le language de programmation fonctionnel avec types dépendants du système Coq. L'implémentation traite le calcul complet des cellules, et sert ainsi de support au programme de décision. Cette implémentation est servie par le système de types dépendants du language, qui permet une implémetation fonctionnelle élégante.

Une bibliothèque certifiée d'arithmétique polynomiale efficace. Une étape préliminaire à l'implémentation d'un tel algorithme est la conception d'une bibliothèque de calcul sur les polynômes adaptée à l'exécution à l'intérieur du système Coq. La forme de Horner creuse permet de réaliser une telle bibliothèque, comprenant les opérations d'anneaux, des opérations formelles comme la dérivée ou la troncature, une division euclidienne et le calcul du plus grand commun diviseur par l'algorithme des sous-résultants.

Une tactique pour la décision dans les structures d'anneaux. Cette représentation efficace est utilisée pour le calcul de forme normale de termes dans une structure d'anneau (ou de semi-anneau) dans une tactique réflexive d'automatisation des preuves d'égalités dans les anneaux/semi-anneaux. Cette tactique améliore la version disponible dans la distribution standard du système selon deux directions orthogonales : l'une d'elle est la représentation des polynômes de la forme abstraite, qui utilise la forme de Horner creuse, l'autre est d'exploiter au mieux les possibilités de calcul sur les coefficients lors de la syntaxification des termes, pour rendre le calcul de forme normale efficace.

Une preuve formelle de l'algorithme des sous-résultants. La preuve de l'algorithme de calcul de pgcd implémenté dans la bibliothèque est une preuve de calcul formel très technique qui met en jeu des notions d'algèbre multilinéaire, au travers d'une notion de déterminant polynomial. La formalisation de cette preuve a ainsi conduit à proposer une définition en Coq des déterminants, par une règle de calcul, qui permet de travailler rapidement avec des applications multilinéaires, antisymétriques et alternées, dont les coordonnées des arguments peuvent être dans un anneau, sans avoir besoin de construire la hiérarchie des structures algébriques jusqu'aux modules de type fini et à l'algèbre multilinéaire.

État d'avancement de la preuve formelle. Le bilan à ce jour des preuves formelles achevées est le suivant. La *structure d'anneau* des polynômes en forme de Horner creuse est établie, grâce à un travail commun avec L. Rideau. La preuve de *l'algorithme de division* euclidienne a été terminée par L. Pottier, reste à prouver l'inégalité des degrés. Sous cette hypothèse, qui reste en axiome, la preuve de l'algorithme de calcul de pgcd par *l'algorithme des sous-résultants* est en grande partie terminée : la preuve du lemme de décalage est terminée et celle du théorème fondamental est très proche d'être achevée : il ne reste plus qu'à régler quelques problèmes causés par les difficultés de réécriture dans les sétoïdes. Ce dernier travail a bénéficié des suggestions de L. Théry pour la formalisation des déterminants et de l'aide de L. Rideau pour les preuves formelles dans les déterminants. Y. Bertot a achevé récemment la preuve de la version faible de la *loi de Descartes* qui assure la correction du procédé d'isolation des racines réelles d'un polynôme. Cette dernière preuve utilise les rationnels de la bibliothèque standard pour évaluer les théorèmes à fournir à partir de l'axiomatique donnée pour les structures de coefficients rationnels. F. Guilhot a proposé une preuve de *correction du procédé de dichotomie* en utilisant la bibliothèque standard des réels et en paramétrant son développement par un axiome de spécification de la fonction de calcul des coefficients de Bernstein.

Perspectives

Architecture de la preuve formelle complète de correction.

L'achèvement de la preuve complète de la correction de l'algorithme de CAD constitue

à la fois un défi et un test de grande échelle pour le système de preuve formelle. Les preuves de correction de l'opérateur de projection constituent par exemple une étape significative du développement restant. En effet, la correction de ces opérateurs de projection repose sur les propriétés de la clôture algébrique du corps réel clos, et sur la continuité des racines d'un polynôme. Cette preuve du théorème de correction de la tactique réflexive va dépendre à la fois de la robustesse de l'architecture choisie pour le programme et de la souplesse du système qui doit permettre à l'utilisateur de programmer aisément des outils de preuve *ad hoc* (simplification d'expressions algébriques, réécriture).

Heuristiques. La tactique reposant sur l'algorithme de CAD programmé est complète, même si cette propriété n'est pas prouvée formellement puisqu'elle est indépendante de la correction de la procédure de décision. La conception d'un outil complet présente un intérêt intrinsèque, tant théorique que de test des limites du système de preuve. Néanmoins il existe de nombreux cas dans lesquels le calcul de CAD est trop coûteux et peut être court-circuité par une heuristique efficace. L'étude des algorithmes heuristiques pour les problèmes d'inégalités polynomiales réelles, celles utilisées par exemple par un système comme Mathematica, devrait apporter des techniques élégantes, efficaces, et si possible moins délicates à prouver formellement, pour construire une batterie d'outils de semi-décision rapides au dessus de la tactique complète basée sur la CAD.

Modules. Dans ce développement, le système actuel de modules de Coq s'est avéré très adapté à l'abstraction de l'implémentation des coefficients rationnels mais il devient inutilisable du fait de la structure du programme lorsqu'il s'agit de partager du code entre cas de base et cas inductif. Le système Coq propose dans son état actuel deux mécanismes d'abstractions, sections et modules, qui coexistent sans être complètement compatibles. Pouvoir combiner ces deux outils est un problème délicat et repose sur des choix sémantiques mais dans un exemple comme celui-ci il sera intéressant d'explorer ces pistes qui apporteraient un confort significatif à la programmation et surtout à la preuve.

Automatisation en géométrie. La formalisation du raisonnement géométrique a donné lieu à plusieurs travaux visant à proposer un cadre métathéorique au raisonnement géométrique. Dans sa thèse (Narboux 2006), J. Narboux a proposé une formalisation en Coq de l'axiomatique de Tarski ainsi qu'une axiomatique de la géométrie plane sur laquelle il implémente une procédure de décision basée sur la méthode des aires. Cette procédure de décision transforme d'abord le problème géométrique en un ensemble d'équations, puis, elle résout le système obtenu. La combinaison de cette procédure de décision avec une tactique comme celle décrite dans ce travail permettrait de fournir une procédure très riche pour l'automatisation du raisonnement géométrique en autorisant des hypothèse d'inégalité dans les systèmes. Ceci augmente considérablement les problèmes de géométrie que l'on peut automatiser. D'autres procédures de décisions pour la géométrie deviennent également accessibles à partir de la bibliothèques d'arithmétique polynomiale développée dans ce travail, en particulier grâce aux algorithmes de divisions/pseudo-divisions, comme la méthode de Wu (Wu 1978).

Références

- Allen, S. F., R. L. Constable, D. J. Howe, et W. Aitken (1990). The Semantics of Reflected Proof. In *Proceedings of the 5th Symposium on Logic in Computer Science*, Philadelphia, Pennsylvania, pp. 95–197. IEEE : IEEE Computer Society Press.
- Artin, E. et O. Schreier (1927). Algebraische konstruktion reeller körper. *Abhandlungen Math. Sem. Univ. Hamburg 5*, 85–99. Traduction en français : `http://www.math.univ-rennes1.fr/geomreel/seminaire_fichiers/Artin-Schreier.pdf`.
- Aubry, P., F. Rouillier, et M. Safey (2002). Real solving for positive dimensional systems. *Journal of Symbolic Computation 34* (6), 543–560.
- Avigad, J., K. Donnelly, D. Gray, et P. Raff (2005). A formally verified proof of the prime number theorem. To appear in ACM Transactions on Computational Logic.
- Balaa, A. et Y. Bertot (2002). Fonctions récursives générales par itération en théorie des types. In *Actes des Journées francophones des langages applicatifs*. INRIA. `http://jfla.inria.fr/2002/index.html`.
- Ballarin, C. (1999). Computer algebra and theorem proving. Rapport technique 473, University of Cambridge.
- Barthe, G., J. Forest, D. Pichardie, et V. Rusu (2006). Defining and reasoning about recursive functions : A practical tool for the coq proof assistant. In *FLOPS*, pp. 114–129.
- Basu, S., R. Pollack, et M.-F. Roy (2003). *Algorithms in Real Algebraic Geometry*, Volume 10 de *Algorithms and Computation in Mathematics*. Berlin Heidelberg New York : Springer.
- Basu, S., R. Pollack, et M.-F. Roy (2006). *Algorithms in Real Algebraic Geometry* (second ed.), Volume 10 de *Algorithms and Computation in Mathematics*. Berlin Heidelberg New York : Springer. //http ://perso.univ-rennes1.fr/marie-francoise.roy/bpr-posted1.html .
- Bertot, Y. (2006). Affine functions and series with co-inductive real numbers. *Mathematical Structures in Computer Science*. Cambridge University Press.
- Bertot, Y. et P. Casteran (2004). *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag.
- Bishop, E. (1967). *Foundations of Constructive Analysis*. McGraw-Hill.
- Bochnak, J., M. Coste, et M.-F. Roy (1998). *Real Algebraic Geometry* (second ed.), Volume 36 de *Ergebnisse der Mat.* Berlin Heidelberg New York : Springer.
- Boutin, S. (1997). Using reflection to build efficient and certified decision procedures. In *TACS*, pp. 515–529.

- Brown, C. W. (2003). QEPCADB - A program for computing with semi-algebraic sets using CADs. *ACM SIGSAM Bulletin 37*(4), 97–108.
- Brown, C. W. (2004). Tutorial Cylindrical Algebraic Decomposition. In *Presented at ISSAC 2004*.
- C-CoRN (2002). A Constructive Coq Repository at Nijmegen. `http://c-corn.cs.ru.nl/`.
- Chambert-Loir, A. (2006). Algèbre commutative, Master 1. `http://name.math.univ-rennes1.fr/antoine.chambert-loir`.
- Chicli, L., L. Pottier, et C. Simpson (2002). Mathematical quotients and quotient types in coq. In H. Geuvers et F. Wiedijk (Eds.), *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Selected Papers*, pp. 95–107.
- Chrzaszcz, J. (2003). *Modules in Type Theory with Generative Definitions*. Thèse de doctorat, Warsaw University et Université Paris-Sud.
- Ciaffaglione, A. et P. D. Gianantonio (2000). A tour with constructive real numbers. In *Proceedings of the workshop Types for Proofs and Programs*, Volume 2277 de *LNCS*, pp. 41–52.
- Collins, G. (1975). Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. *LNCS 33*, 134–183.
- Collins, G. E. (1967). Subresultant and Reduced Polynomial Remainder Sequences. *Journal of the ACM 14*, 128–142.
- Constable, R. L., S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, et S. F. Smith (1986). *Implementing Mathematics with the Nuprl Development System*. NJ : Prentice-Hall.
- Coq development team (2004). *The Coq Proof Assistant Reference Manual, version 8.0*. Coq development team.
- Coquand, T. (1991). Constructive topology and combinatorics. In *Constructivity in Computer Science, Summer Symposium, San Antonio, Texas, USA, June 19-22, Proceedings*, Volume 613 de *Lecture Notes in Computer Science*, pp. 159–164. Springer.
- Coquand, T. (1997). A note on the open induction principle. http ://www.cs.chalmers.se/coquand/intuitionnism.html.
- Coquand, T. et G. Huet (1988). The Calculus of Constructions. *Information and Computation 17*(2/3), 95–120.
- Coquand, T. et C. Paulin-Mohring (1990). Inductively defined types. In P. Martin-Löf et G. Mints (Eds.), *Proceedings of Colog'88*, Volume 417 de *Lecture Notes in Computer Sciences*. Springer.
- Corvez, S. (2005). *Etude de systèmes polynomiaux : contributions à la classification d'une famille de manipulateurs et au calcul des intersections de courbes A-splines*. Thèse de doctorat, Université de Rennes 1.
- Cruz-Filipe, L., H. Geuvers, et F. Wiedijk (2004). C-CoRN, the Constructive Coq Repository at Nijmegen. In A. T. Andrea Asperti, Grzegorz Bancerek (Ed.), *Mathematical Knowledge Management, Proceedings of MKM 2004*, Volume 3119 de *Lecture Notes in Computer Sciences*, Berlin Heidelberg New York, pp. 88–103. Springer.

- Cruz-Filipe, L. et P. Letouzey (2006). A large-scale experiment in executing extracted programs. *Electr. Notes Theor. Comput. Sci. 151*(1), 75–91.
- de Bruijn, N. G. (1970). The mathematical language AUTOMATH, its usage and some of its extensions. In *Symposium on Automatic Demonstration*, Volume 125 de *Lecture Notes in Mathemtics*, pp. 29–62. Springer.
- Delahaye, D. (2000). A Tactic Language for the System `Coq`. In *LPAR, Reunion Island*, Volume 1955, pp. 85–95. Springer-Verlag LNCS/LNAI. `http://cedric.cnam.fr/~delahaye/publications/LPAR2000-ltac.ps.gz`.
- Delahaye, D. et M. Mayero (2001). `Field` : une procédure de décision pour les nombres réels en `Coq`. In *Journees Francophones des Langages Applicatifs, Pontarlier*. INRIA. `http://cedric.cnam.fr/~delahaye/publications/JFLA2000-Field.ps.gz`.
- Delahaye, D. et M. Mayero (2005). Quantifier Elimination over Algebraically Closed Fields in a Proof Assistant using a Computer Algebra System. In *Proceedings of Calculemus 2005*.
- Dolzmann, A., A. Seidl, et T. Sturm (2004). Efficient projection orders for cad. In J. Gutierrez (Ed.), *Symbolic and Algebraic Computation, International Symposium ISSAC 2004, Santander, Spain, July 4-7, 2004, Proceedings*. ACM.
- Dos Reis, G., B. Mourrain, R. Rouillier, et P. Trébuchet (2002). An environment for symbolic and numeric computation. In *Proceedings of the International Conference on Mathematical Software, World Scientific*, pp. 239–249. Le manuel de reférence est disponible à l'adresse `http://www-sop.inria.fr/galaad/logiciels/synaps/documentation.html`.
- Ducos, L. (2000). Optimizations of the subresultant algorithms. *Journal of Pure and Applied Algebra 145*, 149–163.
- Fridlender, D. (1999). An interpretation of the fan theorem in type theory. In *Types for Proofs and Programs, International Workshop TYPES'98. Selected Papers.*, Volume 1657 de *Lecture Notes in Computer Science*, pp. 93–105. Springer-Verlag.
- G. Barthe, V. C. et O. Pons. (2003). Setoids in type theory. *Journal of Functional Programming 13(2)*, 261–293.
- Gabbay, D. M. (1973). The Undecidability of Intuitionistic Theories of Algebraically Closed Fields and Real Closed Fields. *J. Symb. Log. 38*(1), 86–92.
- Geddes, K., S. R. Czapor, et G. Labahn (1992). *Algorithms for Computer Algebra*. Kluwer Academic Publishers.
- Geuvers, H. et M. Niqui (2002). Constructive reals in Coq : Axioms and categoricity. In P. Callaghan, Z. Luo, J. McKinna, et R. Pollack (Eds.), *Types for Proofs and Programs : International Workshop, TYPES 2000, Durham, UK, December 8–12, 2000. Selected Papers*, Volume 2277 de *Lecture Notes in Computer Science*, pp. 79–95. Springer-Verlag.
- Geuvers, H., R. Pollack, F. Wiedijk, et J. Zwanenburg (2002). A Constructive Algebraic Hierarchy in Coq. In S. Linton et R. Sebasitani (Eds.), *Special Issue on the Integration of Automated Reasoning and Computer Algebra Systems*, Volume 34, pp. 271–286. Elsevier.
- Grégoire, B. et X. Leroy (2002). A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pp. 235–246. ACM Press.

- Grégoire, B. et L. Théry (2006). A Purely Functional Library for Modular Arithmetic and its Application for Certifying Large Prime Numbers. In *Proceedings of IJCAR'06 : Third International Joint Conference on Automated Reasoning*. to appear.
- Griffin, T. (1990). A formulae-as-types notion of control. In *POPL*, pp. 47–58.
- Grigor'ev, D. (1988). The Complexity of deciding Tarski algebra. *Journal of Symbolic Computation 5*, 65–108.
- Grégoire, B. (2003). *Compilation des termes de preuves : un (nouveau) mariage entre Coq et Ocaml*. Thèse de doctorat, Université Paris 7.
- Grégoire, B. et A. Mahboubi (2005). Proving ring equalities done right in coq. In *TPHOLs*, pp. 98–113.
- Harrison, J. (1994). Constructing the real numbers in HOL. *Formal Methods in System Design 5*, 35–59.
- Harrison, J. (1996). HOL Light : A Tutorial Introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, Numéro 1166 in LNCS, pp. 265–269. Springer.
- Harrison, J. (1997). Verifying the Accuracy of Polynomial Approximations in HOL. In E. L. Gunter et A. P. Felty (Eds.), *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ, USA, August 19-22, 1997, Proceedings*, Volume 1275 de *Lecture Notes in Computer Science*, pp. 137–152. Springer.
- Harrison, J. (2002). *HOL-Light 2.20 Reference Manual*. `http://www.cl.cam.ac.uk/jrh13/hol-light/reference_220.pdf` : University of Cambridge, DSTO, SRI International.
- Harrison, J. et L. Théry (1998). A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning 21*, 279–294.
- Harrison, J. R. (2005). A HOL theory of Euclidean space. In J. Hurd et T. Melham (Eds.), *Theorem Proving in Higher Order Logics 18th International Conference, TPHOLs 2005*, Volume 3603 de *Lecture Notes in Computer Science*, Oxford, UK, pp. 114–129. Springer-Verlag.
- Heintz, J., M.-F. Roy, et P. Solernó (1990). Sur la complexité du principe de Tarski-Seidenberg. *Bulletin de la Société Mathématique de France 118*, 101–126.
- Hofmann, M. (1995). A simple model for quotient types. In *TLCA '95*, pp. 216–234.
- Hong, H. (1990). *Improvements in CAD-based Quantifier Elimination*. Thèse de doctorat, The Ohio State University.
- Hong, H., R. Liska, et S. Steinberg (1997). Testing stability by quantifier elimination. *Journal of Symbolic Computation 24*(2), 161–187.
- Hörmander, L. (1983). *Tha Analysis of Linear Partial Differential Operators II*, Volume 257 de *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag.
- Howard, W. A. (1980). The formulae-as-types notion of construction. In J. Selding et J. H. Hindley (Eds.), *To H.B. Curry : Essays on Combinatory Logics, Lambda Calculus and Formalism*, pp. 479–490. Academic Press.
- Howe, D. J. (1986). *Implementing Analysis*. Thèse de doctorat, Cornell University.

- Huet, G. (1997). The zipper data structure. *Journal of Functional Programming* (7), 549–554.
- Jacobi, C. (1836). De Eliminatione Variablilis e Duabus Aequationibus. *J. Reine Angew. Math 15*, 101–124.
- Jones, C. (1991). Completing the rationals and metric spaces in LEGO. In G. Huet, G. Plotkin, et C. Jones (Eds.), *Proceedings of the Second Workshop on Logical Frameworks*, pp. 209–222.
- Knuth, D. (1998). *The Art of Computer Programming, Semi-numerical Algorithms* (3rd ed.), Volume 2. Reading,MA : Addison-Wesley. (1st edition, 1969).
- Lang, S. (1995). *Algebra* (3rd ed.). Addison-Wesley.
- Liao, H.-C. et R. J. Fateman (1995). Evaluation of the Heuristic Polynomial gcd. In *ISSAC '95 : Proceedings of the 1995 international symposium on Symbolic and algebraic computation*, pp. 240–247. ACM Press.
- Mahboubi, A. et L. Pottier (2002). Elimination des quantificateurs sur les réels en Coq. In *Journées francophones des Languages Applicatifs*.
- Matiyasevic (1970). The diophantineness of enumerable sets. *Soviet Mathematical Doklady 191*(11), 279–282.
- Mayero, M. (2001). *Formalisation et automatisation de preuves en analyses réelle et numérique*. Thèse de doctorat, Université Paris VI.
- McLaughlin, S. et J. Harrison (2005). A proof-producing decision procedure for real arithmetic. In R. Nieuwenhuis (Ed.), *CADE-20 : 20th International Conference on Automated Deduction, proceedings*, Volume 3632 de *Lecture Notes in Computer Science*, Tallinn, Estonia, pp. 295–314. Springer-Verlag.
- Mourrain, B., F. Rouillier, et M.-F. Roy (2005). Bernstein's basis and real root isolation. *Mathematical Sciences Research Institute Publications*, 459–478. Cambridge University Press.
- Muñoz, C., R. Butler, et V. Carreño (2001). Formal verification of conflict dectection algorithms. In *Conference on Correct Hardware Design and Verification Methods*, Volume 2144 de *Lecture Notes in Computer Sciences*, pp. 403–417. Technical Report, NASA/TM-2001-210864.
- Narboux, J. (2006). *Formalisation et automatisation du raisonnement géométrique en Coq*. Thèse de doctorat, Université Paris Sud.
- Nipkow, T., L. C. Paulson, et M. Wenzel (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Volume 2283 de *LNCS*. Springer.
- Niqui, M. et Y. Bertot (2004). QArith : Coq formalisation of lazy rational arithmetic. In S. Berardi, M. Coppo, et F. Damiani (Eds.), *Types for Proofs and Programs : International Workshop, TYPES 2003, Torino, Italy, April 30 – May 4, 2003, Revised Selected Papers*, Volume 3085 de *Lecture Notes in Computer Science*, pp. 309–323. Springer-Verlag.
- Palmgren, E. (2002). An intuitionistic axiomatisation of real closed fields. *Mathematical Logic Quarterly 48*(2), 297–299.
- Pottier, L. (1999). Algebra. Coq contribution. http ://coq.inria.fr/contribs/algebra.html.
- Rouillier, F. et P. Zimmermann (2004). Efficient isolation of polynomial's real roots. *J. Comput. Appl. Math. 162*(1), 33–50. Elsevier Science Publishers B. V.

- Sacerdoti Coen, C. (2006). A semi-reflexive tactic for (sub-)equational reasoning. In *Types for Proofs and Programs,2004, Revised Selected Papers*, Volume 3839 de *Lecture Notes in Computer Science*, pp. 98–114. Springer.
- Simpson, C. (2004). Computer Theorem Proving in Mathematics. *Letters in Mathematical Physics 69*(1-3), 287–315.
- Stein, J. (2003). Linalg. Coq contribution. http ://coq.inria.fr/contribs/LinAlg.html.
- Strzebonski, A. (2000). Solving systems of strict polynomial inequalities. *Journal of Symbolic Computation 29*, 471–580.
- Tarski, A. (1951). A Decision Method for Elementary Algbra and Geometry. University of California Press. 2nd edition, revised, by Alfred Tarski with the assistance of J. C. C McKinsey.
- Théry, L. (2001). A Machine-Checked Implementation of Buchberger's Algorithm. *Journal of Automated Reasoning 26*, 107–137.
- Troelstra, A. (Ed.) (1973). *Metamathematical Investigation of Intutitionnistic Arithmetic and Analysis*, Volume 344 de *LNCS*. Springer Verlag.
- Uspensky, J. (1948). *Theory of Equations*. McGraw-Hill Book Company.
- Veldman, W. (2001). Almost the fan theorem. Rapport technique 0113, Department of Mathematics, University of Nijmegen, Toernooiveld, 6525 ED Nijmegen The Netherlands.
- Vincent, A. J. H. (1836). Sur la Résolution des équations numériques. *Journal de Mathématiques Pures et Appliquées*, 341–371.
- von zur Gathen, J. et T. Lücking (2003). Subresultants Revisited. *Theoretical Computer Science 297*(1-3), 199–239.
- Wu, W.-T. (1978). On the decision problem and the mechanization of theorem proving in elementary geometry. *Scientia Sinica 21*, 157–179.
- Yap, C. K. (2000). *Fundamental Problems of Algorithmic Algebra*. Oxford : Oxford University Press.
- Ying, J. Q., L. Xu, et Z. Lin (1999). A Computational Method for Determining Strong Stability of n-D-systems. *Journal of Symbolic Computation 27*, 479–499.
- Zippel, R. (1993). *Effective Polynomial Computation*. Kluwer Academic Publishers.

Résumé

Le logiciel Coq est un assistant à la preuve basé sur le Calcul des Constructions Inductives. Dans cette thèse nous proposons d'améliorer l'automatisation de ce système en le dotant d'une procédure de décision réflexive et complète pour la théorie du premier ordre de l'arithmétique réelle. La théorie des types implémentée par le système Coq comprend un langage fonctionnel typé dans lequel nous avons programmé un algorithme de Décomposition Algébrique Cylindrique (CAD). Cet algorithme calcule une partition de l'espace en cellules semi-algébriques sur lesquelles tous les polynômes d'une famille donnée ont un signe constant et permet ainsi de décider les formules de cette théorie. Il s'agit ensuite de prouver la correction de l'algorithme et de la procédure de décision associée avec l'assistant à la preuve Coq. Ce travail comprend en particulier une librairie d'arithmétique polynomiale certifiée et une partie significative de la preuve formelle de correction de l'algorithme des sous-résultants. Ce dernier algorithme permet de calculer efficacement le plus grand commun diviseur de polynômes à coefficients dans un anneau, en particulier à plusieurs variables. Nous proposons également une tactique réflexive de décision des égalités dans les structures d'anneau et de semi-anneaux qui améliore les performances de l'outil déjà disponible et augmente son spectre d'action en exploitant les possibilités de calcul du système. Dans une dernière partie, nous étudions le contenu calculatoire d'une preuve constructive d'un lemme élémentaire d'analyse réelle, le principe d'induction ouverte.

Mots clés : Formalisation, Coq, Assistants à la preuve, Procédures de décision, Calcul formel, Décomposition Algébrique Cylindrique, Sous-résultants, Induction ouverte.

Abstract

The Coq system is a proof assistant based on the Calculus of Inductive Constructions. In this work, we propose to enhance the automation of this system by providing a reflexive and complete decision procedure for the first order theory of real numbers. The Type Theory implemented by the Coq system comprises a typed functional programming language, which we have used to implement a Cylindrical Algebraic Decomposition algorithm (CAD). This algorithm computes a partition of the space into sign-invariant, semi-algebraic cells for the polynomials of a given family. Hence it allows to decide all the formulae of this theory. Then we have to prove formally the correctness of the algorithm and of the related decision procedure, using the Coq proof assistant. This work includes a library for certified polynomial arithmetic and a significant part of the formal proof of correctness of the sub-resultants algorithm. This last algorithm allows to compute efficiently the greatest common divisor of polynomials with coefficients in a ring, and in particular of multivariate polynomials. We also propose a reflexive tactic for deciding equalities in ring and semi-ring structures, which enhances the performances of the tool previously available in the system by taking benefit of the computational abilities of the system. In a last part, we study the computational content of a constructive proof of an elementary lemma of real analysis, called principle of open induction.

Key words : Formalization, Coq, Proof assistants, Decision procedures, Computer algebra, Cylindrical Algebraic Decomposition, Subresultants, Open induction.